Konttiteknologioiden käyttö verkkosivustojen kehittämisessä ja ylläpidossa

Pro Gradu -tutkielma
Turun yliopisto
Tulevaisuuden teknologioiden laitos
Tietojenkäsittelytiede
2018
Roope Merikukka

Tarkastajat:
Ville Leppänen
Johannes Holvitie

TURUN YLIOPISTO
Tulevaisuuden teknologioiden laitos

ROOPE MERIKUKKA: Konttiteknologioiden käyttö verkkosivustojen kehittämisessä ja ylläpidossa

Pro Gradu -tutkielma, 77 s., 3 liites.
Tietojenkäsittelytiede
Maaliskuu 2018

Verkkosivustojen määrä internetissä kasvaa alati. Myös sivustoja tuottavia yrityksiä ilmestyy markkinoille jatkuvasti etsien uusia ja kilpailukykyisiä tapoja toteuttaa asiakkaidensa vaatimukset.
Konttiteknologioiden rooli verkkosivustojen ja -sovellusten kehityksessä on kasvava trendi. Konttiteknologiat mahdollistavat sovellusten eristämisen toisistaan, jolloin sovelluksia voidaan käynnistää, monistaa sekä testata entistä helpommin ja turvallisemmin.
Docker on yksi konttiteknologioiden käytetyimmistä sovelluksista ja sen pohjana on tuotettavien sovellusten uudelleenkäytettävyys, eristäminen sekä skaalautuvuus. Sillä on tärkeä rooli sovellusten paketoinnin standardisoijana sekä standardisoituna käyttöliittymänä näiden pakettien suorittamiselle.
Esittelen tässä tutkielmassani verkkosivustojen toteutuksen työvaiheet. Esittelen myös konttiteknologiat, ensin yleisesti, jonka jälkeen syvennyn konttiteknologioista Dockeriin ja sen toteutukseen sekä ominaisuuksiin. Tämän jälkeen tutkin millaisia erilaisia rooleja Dockerilla on ja miten näitä voi käyttää hyväksi sovelluskehityksen tukena. Tätä tietoa käytän tapaustutkimuksessa, jossa toteutan Dockeria käyttävän sovelluskehityspohjan niin kehittämisen kuin ylläpidon tueksi tapaustutkimuksen kohdeyritykselle. Lopuksi analysoin tutkimuksen tulokset, ja esittelen omat johtopäätökseni Dockerin käytöstä kehityksen apuna sekä tapaustutkimuksen kannalta.

Asiasanat: docker, konttiteknologia, verkkosivu, sovelluskehitys

UNIVERSITY OF TURKU
Department of Future Technologies

ROOPE MERIKUKKA: Container technologies in web development and production

Master's thesis, 77 p., 3 app. p.
Computer Science
March 2018

The amount of websites grows continuously with increasing speed. At the same time new web development companies pop up to find competitive ways to implement their clients' needs.
The role of container technologies in web and software development is a growing trend. Containers enable isolation of applications from each other and so applications can be started, duplicated and tested easier and safer.
Docker is one of the most used applications in the container market. It's base idea lies on application reusability, isolation and scalability. Docker also has a major role as a standard for packaging software and as an API to run packaged software.
In this thesis I present the main steps of web development process. I also present container technologies in general. Then I give an in depth introduction to Docker, it's implementation, functionalities and features. After that I research the different roles of Docker and how those can be used to support software development. I will use this information in my case study where I use Docker to implement web development boilerplate for both development and production use for the target company. Finally I analyze the case study results and present my own conclusions of Docker usage as a supportive technology from both perspectives; the case study and in general.

Keywords: docker, container technology, website, software development

Sisältö

Kuvat

Koodiesimerkit

Luku 1

Johdanto

Konttiteknologiat ovat alati yleistyvä väline niin sovellusten tehokkaaseen eristämiseen kuin niiden helppoon siirrettävyyteen, uudelleenkäytettävyyteen ja skaalaamiseen. Konttiteknologioilla tarkoitetaan kokoelmaa teknologisia ratkaisuja, joilla mahdollistetaan sovelluksen eristäminen omalle tasolleen siten, että ne eivät oletusarvoisesti ole tietoisia ympärillään olevista muista sovelluksista ja niiden käyttöoikeudet ovat melko tiukasti rajatut. Konttiteknologiat mahdollistavat myös kehittäjien ja palvelinylläpitäjien välisen rajapinnan supistamisen.

Tutkielmani on jaettu tämän johdannon lisäksi viiteen erilliseen lukuun. Toisessa luvussa pohdin verkkosovellusten tuotantoprosessia ja käyn läpi verkkosivustojen toteutuksen työvaiheet ja hieman niiden sisältöä.

Kolmannessa luvussa käyn läpi konttiteknologioita. Ensin käyn läpi taustoja ja tarpeita konttiteknologioiden kehittämiselle sekä hieman konttiteknologioiden historiaa. Tämän jälkeen käyn lyhyesti läpi konttiteknologioiden arkkitehtuuria ja lopuksi pureudun tällä hetkellä käytetyimpään konttiteknologiaan, eli Dockeriin. Käyn läpi Dockerin teknisen rakenteen, sen toimintaa sekä lyhyesti myös Dockerin käyttöönoton.

Neljännessä luvussa tutkin miten Docker toimii osana verkkosivustojen ja -sovellusten kehitystä. Käyn ensin läpi teemaa yleisesti, jonka jälkeen pohdin Dockerin rooleja osana sovelluskehityksen eri vaiheita. Lopuksi tutkin miten Docker on vaikuttanut nykyaikai-

seen verkkosovelluskehitykseen.

Viides luku on tapaustutkimus, jonka tein pienessä verkkosivustoja suunnittelevassa ja toteuttavassa yrityksessä. Käyn tapaustutkimuksessani ensin läpi kohdeyrityksen teknologian ja toimintatavat ennen tutkimusta, jonka jälkeen käyn läpi konttiteknologioiden käyttöönoton. Olen jakanut käyttöönoton kolmeen osaan: suunnitelmaan, toteutukseen ja viimeisenä analyysiin, jossa käyn läpi miten käyttöönotto onnistui.

Viimeisessä luvussa käyn läpi työni keskeisimmät johtopäätökset, sekä mitä työn tuloksista voidaan olettaa ja miten tästä kannattaa jatkaa.

Luku 2

Verkkosivuston toteutuksen työvaiheet

Verkkosivustojen määrä maailmassa kasvaa koko ajan kiihtyvää tahtia [1]. Samalla myös verkkosivustoja toteuttavia yrityksiä syntyy alati, etsien kilpailukykyisiä tapoja toteuttaa asiakkaidensa kasvavat vaatimukset. Vaikka nykyaikaisten verkkosivustojen toteutus vastaa hyvin pitkälti monimutkaista sovelluskehitystä, ei alalla ole toteutuksille täysin vakiintunutta standardia tai metodologiaa käytössä, tai ainakin tätä on tutkittu vähän verrattaen perinteiseen sovelluskehitykseen. Eräs syy tälle saattaa olla se että verkkosovellukset kuitenkin poikkeavat sekä perinteisistä sovelluksista että toisistaan suuresti muun muassa rakenteensa, informaatioarkkitehtuurinsa sekä tarkoituksensa puolesta [2].

Ketterässä sovelluskehitysmallissa tarkoituksena on toimittaa toimivaa koodia useasti. Tämä saavutetaan jakamalla kehitystyö iteraatioihin, joiden lopuksi asiakkaalle toimitetaan eniten lisäarvoa tuottavat ominaisuudet. Kriitikoiden mielestä tämä kuitenkin vähentää merkittävästi esimerkiksi arvokkaan dokumentaation määrää etenkin monimutkaisissa projekteissa [3]. Näistä syistä verkkosivustojen toteutuksen prosessi noudattelee usein jotakin perinteisen vesiputousmallin sekä ketterien toimintatapojen väliltä. Seuraavassa esittämäni työvaiheet pohjautuvat sekä alan yleisiin käytäntöihin että osin luvun 5 tapaustutkimuksen kohdeyrityksen toimintatapoihin.

Verkkosivuston kehittäminen koostuu useista eri työvaiheista, aina sivuston määrittelystä sen julkaisemiseen ja ylläpitoon. Vaiheet ovat periaatteessa toisiaan seuraavia ja

noudattelevat siten jokseenkin vesiputousmallia, eli kun ensimmäinen vaihe on saatu valmiiksi, siirrytään seuraavaan vaiheeseen. Vaiheet voivat toimia myös iteratiivisesti ketterän kehityksen paradigman mukaisesti, jolloin jokainen vaihe toistuu kehityksen aikana useaan kertaan ja yhdessä iteraatiossa käydään läpi jokainen työvaihe. Useasti kehitystyö saattaa tapahtua myös kahden edellisen yhdistelmänä, jolloin asiat etenevät vesiputousmallin mukaisesti, mutta vaiheet kulkevat osittain toisiinsa nähden limittäin ja edellisiin vaiheisiin voidaan helposti palata. Kehitettävällä sivustolla tai sovelluksella on kuitenkin yleensä tietty hetki, jolloin sen katsotaan olevan julkaistu ja tällöin siirrytään julkaisun jälkeiseen aikaan. Verkkosivuston tai sovelluksen toteuttamisen eri vaiheet voidaan jakaa pääpiirteittäin seuraaviin työvaiheisiin:

1. Määrittely
2. Suunnittelu
3. Kehitystyö
4. Testaus
5. Julkaisu
6. Ylläpito & jatkokehitys

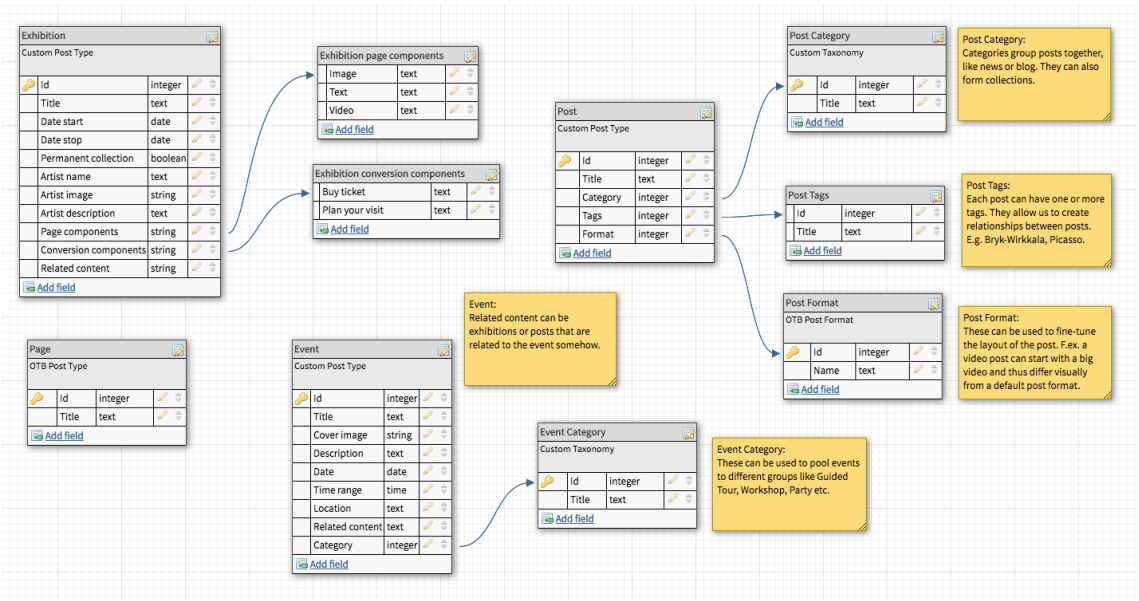
Julkaisun jälkeen voidaan siirtyä takaisin määrittelyvaiheeseen, sillä usein verkkosivustoa ei voida katsoa valmiiksi julkaisun koittaessa, vaan kehittämistyötä jatketaan ylläpidon lomassa koko verkkosivuston tai sovelluksen elinkaaren ajan. Näinpä tämä silmukkarakenne voidaan jakaa karkeasti kahteen erilliseen osaan: ennen ensimmäistä julkaisua tapahtuvaan kehitystyöhön sekä ensimmäisen julkaisun jälkeiseen ylläpitoon ja kehitystyöhön. Käyn läpi seuraavaksi tarkemmin kunkin työvaiheen sisältöä ja ajatuksia sen takana. Tässä tapauksessa teen oletuksen (niiltä osin kuin se on tarpeellista), että kaikki työvaiheet suoritetaan yhden yrityksen toimesta projektin alusta loppuun. Reaalimaailmassa ei toki ole tavatonta, että jotkin työvaiheet ulkoistetaan tai projektia tekee useampi

yritys yhtäaikaisesti. Tällä saatetaan saavuttaa useita hyötyjä kun yritykset voivat näin erikoistua omiin osa-alueisiinsa. Tällaisella toimintatavalla saattaa myös olla haittapuolia. Se saattaa huonoissa kokoonpanoissa heikentää projektin kommunikaatiota ja näin vaikuttaa negatiivisesti esimerkiksi aikatauluihin ja pahimmassa tapauksessa jopa heikentää työn laatua.

2.1 Määrittely

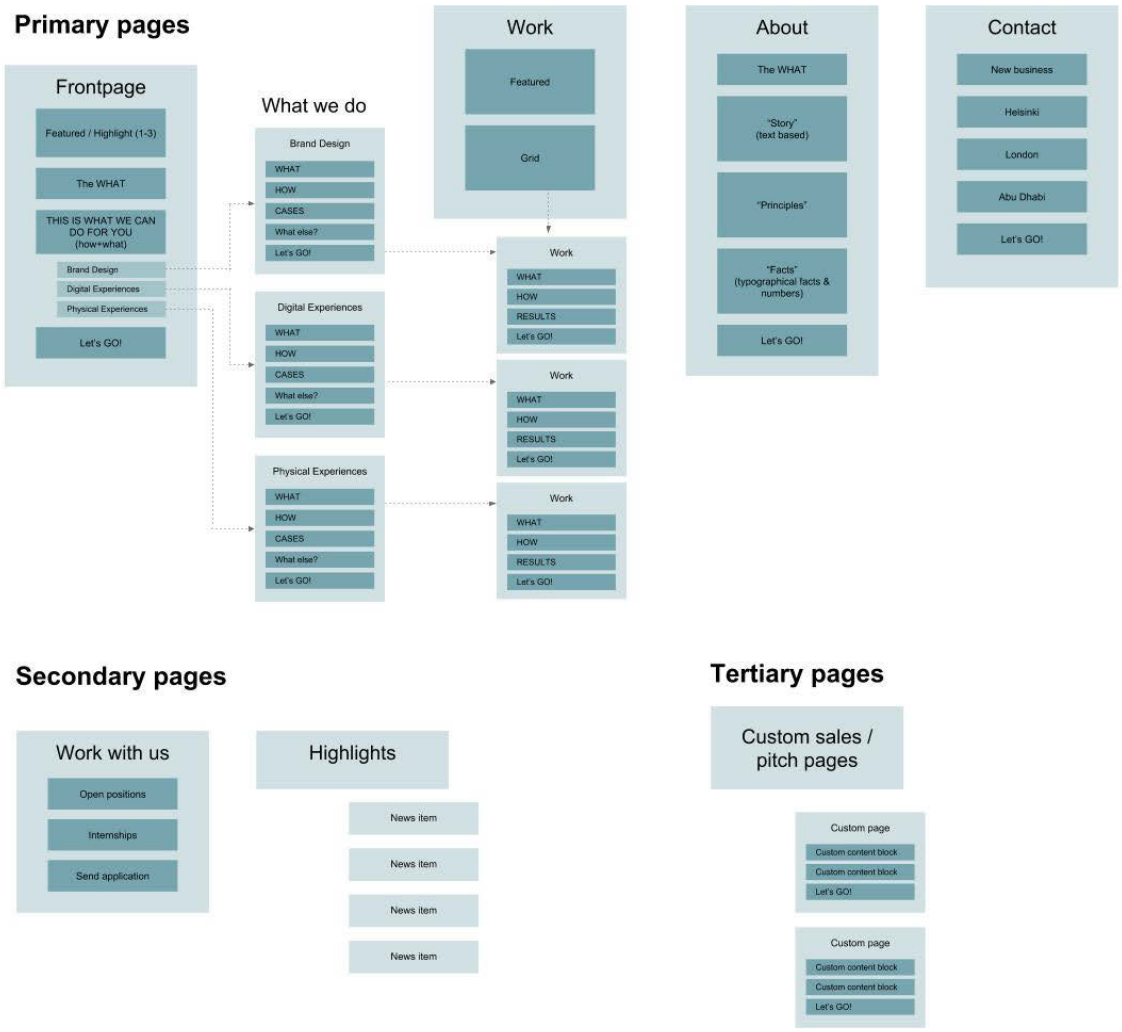
Projektille, oli se sitten verkkosivusto tai mikä tahansa muu sovellus, on tärkeää määritellä raamit, joiden mukaan projektin tulee edetä. Määrittelyvaiheessa tulee saada riittävä ymmärrys projektin tavoitteista, jotta seuraaviin vaiheisiin voidaan edetä.

Kun kyseessä on verkkosivusto, tarkoittaa määrittely esimerkiksi tarvittavien dokumenttien, kuten esimerkiksi informaatioarkkitehtuurin (kuva 2.1), tuottamista. Määrittelyvaiheessa on hyvä ottaa selvää tulevan sivuston sisällöstä ja luoda yksinkertainen sivukartta (kuva 2.2) suunnittelun ja kehityksen tueksi. Lisäksi projektin määrittelyvaiheessa on hyvä selvittää esimerkiksi taustalla käytettävät teknologiset ratkaisut siinä määrin kuin mahdollista, sillä tällä voi olla vaikutusta seuraavissa vaiheissa. Määrittelyvaiheessa selvitetään myös usein projektin vaatimat resurssit, kuten kuinka paljon työvoimaa projektiin tulee kiinnittää.



Kuva 2.1: Esimerkki sivuston informaatioarkkitehtuurista

Yksi tärkeimmistä määrittelyvaiheen aikana tuotettavista dokumenteista on tekninen määrittely, joka sisältää raamit sovelluksen toiminnalle niin funktionaalisesti, kuin bisneslogiikankin kannalta. Tämä dokumentti sisältää mm. projektin tavoitteen, sovelluksen jonkin tasoisen järjestelmäkuvauksen, käyttöliittymäkuvaukset, taustaprosessit, tietokantamallit, mahdolliset integraatiokuvaukset sekä funktionaaliset vaatimukset. Jako voidaan tehdä myös vielä korkeammalla tasolla sekä funktionaalisiin (*functional*) vaatimuksiin, että ei-funktionaalisiin (*non-functional*) vaatimuksiin. Usein tämä dokumentti ei tule valmiiksi vielä määrittelyvaiheen aikana vaan sitä edistetään projektin edetessä. Suurimmat raamit on kuitenkin hyvä määritellä ja pohtia jo heti projektin alkaessa, jotta projekti pysyy sovitussa laajuudessa ja kustannusarviossa.



Kuva 2.2: Esimerkki sivuston sivukartasta

Sovellukselle tai verkkosivustolle valitaan usein niin sanotut KPI:t eli *Key Performance Indicator*it. Nämä ovat laskettavia tai vertailtavia määreitä, joiden avulla asiakas voi seurata esimerkiksi liiketoimintansa tavoitteita. Nämä voivat olla vaikkapa tietyn asiakasmäärän saavuttaminen tai myynnin kasvattaminen. Näiden määrittelyvaiheessa valittavien indikaattoreiden tulisi vaikuttavaa tuleviin työvaiheisiin, kuten suunnitteluun, toimien näiden tukena.

Määrittelyvaihe on yksi projektin tärkeimmistä työvaiheista, sillä sitä käytetään pohjana kaikkiin tuleviin työvaiheisiin. Epäonnistuminen tässä vaiheessa lisää työmäärää huo-

mattavasti, mikäli asioita joudutaan myöhemmin muuttamaan tai korjaamaan huolimattomasti tehdyn määrittelyn vuoksi. Määrittelyllä on suora vaikutus projektin onnistumiselle ja huonosti tehty määrittely voi johtaa koko projektin epäonnistumiseen [4].

2.2 Suunnittelu

Kun määrittely on tehty, voidaan siirtyä suunnittelemaan sovellusta. Kaikki työ voidaan jossain määrin laskea kuuluvan suunnittelun piiriin, sillä onhan esimerkiksi määrittelykin tietynlaista suunnittelua. Tässä kontekstissa suunnittelulla kuitenkin tarkoitan lähinnä visuaalista ja rakenteellista suunnittelua.

Tässä vaiheessa sivustosta rakennetaan yksinkertaisia prototyyppejä, jotka strukturoivat sisältöä sekä esittelevät sivuston tärkeimpiä toiminnallisuuksia. Yksinkertaisimmillaan prototyyppi voi olla paperille tehty luonnos tai erinäisillä työkaluilla tuotettu ns. click-demo, jossa esitetään rakennetta ja toimintaa esimerkiksi kuvien päälle liimatuin toiminnallisuuksin. Eräs työkalu tällaisten prototyyppien tuottamiseen on esimerkiksi InVision.

Suunnittelua jatketaan myös rinnakkain itse kehitystyön aikana, kun suunnittelullisia tarpeita ilmenee sovelluksen kehittyessä. Kehitystyö nostaa usein eteen uusia haasteita ja tarpeita, joita suunnittelussa ei välttämättä ole alussa otettu huomioon. Suunnitellut ominaisuudet ja toimenpiteet eivät välttämättä aina sovellukaan yhteen esimerkiksi puutteellisen datan tai ns. täydelliseksi suunnitellun sisällön takia. Määrittelyvaiheessa valitut teknologiaratkaisut saattavat asettaa suunnittelulle tiettyjä rajoitteita esimerkiksi toiminnallisuuksien puutteiden tai niiden hankalan toteutettavuuden vuoksi.

Suunnittelun kulmakivenä tulisi nykystandardien mukaan pitää aina käytettävyyttä sekä olemassa olevaa tietoa ja analytiikkaa sekä kohderyhmästä kenelle sovellusta ollaan tekemässä, että mahdollisen aiemman sovelluksen tai sivuston suorituskyvystä. Näin pystytään helposti ohittamaan ilmeisimmät sudenkuopat. Suunnittelutyö tarvitsee siis lähes

aina dataa tuekseen ja tämä data pitää muotoilla sovellukseksi; ”Design needs data and data needs design.”

Suunnittelijan tuleekin ymmärtää sekä markkinoinnillista puolta että teknisiä mahdollisuuksia ja rajoitteita. Mikäli verkkosivusto suorittaa bisneslogiikkaa, tulisi suurimman osan suunnitelluista malleista tukea tätä ulkonäköseikkojen sijaan. [2]

2.3 Kehitystyö

Sovelluskehitys on työvaiheista usein laajin, ja se tapahtuu rinnakkain niin suunnittelun kuin testauksenkin kanssa. Nykyään on tavanomaista, että kehitystä ei aloiteta tyhjästä, vaan apuna ja alustana käytetään jotakin sovelluskehystä (*framework*) tai verkkosivuston tapauksessa julkaisujärjestelmää kuten WordPressiä. Myös mahdollisesti aiemmissa projekteissa tehtyjä komponentteja ja konventioita uudelleenkäytetään. Tämä säästää niin työtunneissa (tai jopa kuukausissa) kuin turhassa pään hakkaamisessa seinään kun pyörää ei tarvitse keksiä uudestaan. Useimmiten myös esimerkiksi avoimen lähdekoodin sovelluskehyksissä ei itse tarvitse pitää huolta mahdollisista tietoturvaongelmista, vaan ne korjataan suhteellisen nopeasti yhteisön toimesta.

2.4 Testaus

Lopullinen tuotettu sovellus tulee aina testata ennen sen julkaisemista. Verkkosivustojen tapauksessa tämä tarkoittaa usein muutamaa erilaista testausmenettelyä: selaintestausta, yksikkötestausta sekä integraatiotestausta.

Selaintestaus tarkoittaa nimensä mukaisesti sitä, että sivustoa testataan eri selaimilla, selainversioilla ja eri käyttöjärjestelmillä. Selainvalmistajat toteuttavat usein tietyt ominaisuudet hieman eri tavoin ja osa käytettävistä ominaisuuksista ei välttämättä vielä kuulu viralliseen HTML-standardiin, jolloin selaimet esimerkiksi käyttävät CSS-muotoiluissa prefix-etuliitteitä merkitsemään näitä ominaisuuksia. Selaintestaukseen voidaan käyttää

siihen tarkoitettua laboratoriota, josta löytyy fyysiset tietokoneet ja laitteet, joihin on asennettu eri käyttöjärjestelmiä ja selaimia. Tähän tarkoitukseen löytyy nykyään myös verkkopalveluita, kuten BrowserStack[1].

Yksikkötestauksella testataan lähinnä verkkosivustossa tai -sovelluksessa käytettyjä funktioita ja funktionaalisuuksia. Tämä tarkoittaa sitä, että kirjoitetaan testit, jotka ajavat funktioita erilaisilla syötearvoilla ja testaavat, että tulos on aina haluttu ja toimiva (tai vaihtoehtoisesti antaa oikeanlaisen virheen). Yksinkertaisimmillaan tämä tarkoittaa esimerkiksi sitä, että kun sivustolle on toteutettu funktio, jonka tehtävänä on vaikkapa tarkistaa lomakkeen kenttä, kirjoitetaan testi, joka testaa lomaketta arvoilla, jotka ovat sekä oikein että väärin. Tämä testi ajetaan aina kun lomakkeeseen tehdään muutoksia, jolloin huomataan mikäli muutos on tuonut ei-haluttuja sivuvaikutuksia.

Integraatiotestauksella tarkoitetaan testejä, jotka kirjotetaan lähinnä itse sovelluksen että sen käyttämien rajapintojen välille. Näin pystytään takaamaan, että esimerkiksi rajapintojen virhetilanteista pystytään selviämään.

2.5 Julkaisu

Sivuston tai sovelluksen julkaisu voidaan yleisesti jakaa kahteen erilaiseen tapaan: normaali julkaisu sekä ns. kevyt julkaisu (*soft launch*). Normaalilla julkaisulla tarkoitetaan sitä, että sovellus tuotaan samalla kertaa kaikkien saataville. Kevyt julkaisu eroaa tästä siis siten, että sovellusta ns. beta-testataan ensin pienemmällä yleisömäärällä. Tämä on tavanomaisempaa esimerkiksi kun luodaan pelejä ja tuotetta halutaan vielä testata, ennen kuin se halutaan laajempaan levitykseen. Myös suuremmissa verkkosovelluksissa käytäntö voi olla hyvä, sillä se yleensä antaa kehittäjille mahdollisuuden korjata viimeisimpiä virheitä, sekä tarkistaa esimerkiksi käytettävyyttä vielä ennen varsinaista julkaisua.

[1] https://www.browserstack.com

2.6 Ylläpito & jatkokehitys

Kuten aiemmin jo mainitsinkin, sivusto tai sovellus tarvitsee lähes poikkeuksetta jonkinlaista ylläpitoa ja jatkokehitystä. Verkkosivua voi harvoin kutsua kokonaan valmiiksi. Tämä johtuu esimerkiksi selainten nopeasta kehittymisestä ja siitä, että sovelluksen kehittämiseen käytetään usein ominaisuuksia, jotka eivät vielä kuulu standardien piiriin vaan ovat ns. kokeellisia (*experimental*). Ylläpito taas pitää sisällään esimerkiksi palvelinten huoltoa ja päivitysten asentamista, jotta löydetyt haavoittuvuudet esimerkiksi käytetyissä sovelluskehyksissä tai ohjelmointikielissä tulee paikattua.

2.6.1 Jaettu palvelintila

Jaettu palvelintila on yleensä halvin ylläpitoratkaisu. Sillä tarkoitetaan palvelinta, jossa palvelimen ylläpitäjä jakaa palvelimen useille, usein jopa tuhansille, eri käyttäjille. Käyttäjä siis ”vuokraa oikeuden” palvelimen käyttöön. Ongelmana tämän kaltaisessa ylläpitoratkaisussa on se, että kaikki käyttäjät jakavat yhdessä palvelimen resurssit. Käyttäjä ei myöskään pääse juurikaan konfiguroimaan omaa sivutilaansa, vaan konfiguraatio on yleensä kompromissi. [5]

2.6.2 Virtuaalipalvelin

Virtuaalipalvelin (*VPS, Virtual Private Server*) on seuraava askel jaetusta palvelintilasta ja onkin hyvin balanssissa sekä konfiguroitavuuden että kustannusten suhteen. Virtuaalipalvelin on myös yksi jaetun palvelintilan muoto, mutta tällöin palvelin on jaettu vain muutamien käyttäjien kesken ja siinä hypervisor-niminen alusta pystyttää fyysiselle palvelimelle oman virtuaalisen palvelimen. Tällöin palvelimen resurssit on jaettu siten, että kullakin virtuaalipalvelimen omistajalla on käytössään tietty määrä resursseja, eikä muiden käyttäjien toimet vaikuta näihin. [5]

Virtuaalipalvelinten hyvä puoli on niiden helppo skaalautuvuus. Koska resurssit ovat

jaettu virtuaalikoneille ohjelmallisesti, voidaan resursseja yleensä lisätä tai vähentää helposti ja nopeasti. [5]

2.6.3 Dedikoitu palvelin

Dedikoitu palvelin tarkoittaa ratkaisua, jossa käyttäjä vuokraa palvelintarjoajalta koko tietokoneen itselleen. Näin palvelimesta saadaan kaikkein eniten tehoja irti. Tällöin asiakas saa palvelimen käyttöönsä, mutta palveluntarjoaja on loppupeleissä vastuussa palvelimen raudan toimivuudesta.

2.6.4 Pilvipalvelu

Pilvipalvelut ovat usein pohjimmiltaan virtuaalipalvelimia, mutta pilvipalveluiden tarjoajat eivät halua kutsua palveluaan tällä nimellä. Pilvipalvelut voivat olla myös konttiteknologiaan pohjautuvia tai tilattomia ylläpitoratkaisuja eli ne vaativat sovelluksilta ns. serverless-sovelluskehyksen mukaista toteutusta.

Luku 3

Konttiteknologiat

3.1 Tausta ja motivaatiot

Käyttöoikeuksien eristäminen ja resurssien jakaminen sekä kontrollointi sovellusten kesken on perinteisesti saavutettu palvelinympäristöissä virtuaalikoneiden avulla. Nämä ovat ratkaisevia vaatimuksia, kun palvelimessa ajettavien sovellusten ja prosessien ei voida sallia päästä sekä toistensa tiedostorakenteisiin tai resursseihin. Virtualisoinnin ja konttiteknologioiden hyötyjä ei yleisesti ottaen ole vaikea hakea. Hyödyiksi voidaan määritellä esimerkiksi halpa hinta, kun verrataan vaikkapa virtuaalipalvelimien käyttöä useiden fyysisten laitteiden sijaan. Virtuaalikoneet ovat kuitenkin raskaita, koska ne vaativat laitteiston virtualisoinnin sekä tämän päällä ajettavan käyttöjärjestelmän ja tuovat siis ylimääräisen abstraktiotason. Tämä kuorma joudutaan maksamaan suorituskyvyssä. Konttiteknologiat ovat mielenkiintoinen ja usein huomattavasti vielä virtuaalikoneitakin kevyempi vaihtoehto. Vaikka teknologia ja konseptit konttipohjaisiin rakenteisiin on ollut olemassa jo pitkään, on nämä vasta viime aikoina tuotu valtavirran tietoisuuteen ja käyttöön helppokäyttöisten sovellusten, kuten Dockerin avulla. [6]

3.1.1 Virtualisoinnin tarpeet

Virtualisointi tuo ratkaisun useisiin Unix-pohjaisten käyttöjärjestelmien perinteisiin ongelmiin, joista esimerkkeinä prosessien ajaminen samassa käyttäjätilassa, resurssien allokoinnin puute, sekä koko järjestelmän laajuiset jaetut ohjelmistoriippuvuudet. Prosessien ajamisella samassa tilassa tarkoitetaan sitä, että jokainen prosessi on tietoinen toisista prosesseista ja pystyy omien oikeuksiensa puitteissa puuttumaan muiden prosessien toimintaan. Virtualisointi ratkaisee tämän ongelman eristämällä tarvittaessa prosessit omiksi yksiköikseen ajettavaksi joko virtuaalikoneen tai kontin sisällä. Resurssien allokoinnin puute mahdollistaa esimerkiksi yhden prosessin omivan kaiken vapaana olevan laskenta- tai muun resurssin, jolloin muut prosessit eivät toimi toivotulla tavalla. Virtualisoinnissa jokaiselle yksikölle voidaan määritellä tietty osa isäntäkoneen resursseista käyttöön. Jaettujen ohjelmistoriippuvuuksien ongelma koskee etenkin silloin, jos useat eri sovellukset tai prosessit tarvitsevat toimiakseen jonkin tietyn ohjelmiston eri version. Virtualisointi mahdollistaa eri ohjelmistoversioiden asentamisen erillisten konttien tai virtuaalikoneiden sisälle, jolloin ongelma voidaan poistaa tai sitä voidaan ainakin rajata. [6] Myös niin sanottu ”works on my machine” –ongelma voidaan ratkaista konttiteknologioiden avulla. Tällä ongelmalla tarkoitetaan tilannetta, jossa esimerkiksi tuotantoympäristössä havaitaan jokin virhe, jota ei saada kuitenkaan toistettua testiympäristössä tai kehittäjän omassa ympäristössä. Tällainen ongelma saattaa syntyä testattuun ja toimivaksi todettuun sovellukseen esimerkiksi järjestelmäpäivitysten takia tai siitä syystä, että järjestelmät ovat asentaneet riippuvuuksia eri järjestyksessä toisiinsa verrattuna.

3.1.2 Historiaa

Vuonna 1979 esiteltiin Unix-käyttöjärjestelmän versio 7:ssä uutena ominaisuutena `chroot`-komento, jonka avulla mahdollistettiin prosessien eristäminen toisistaan muuttamalla prosessin root-hakemisto sekä prosessille itselleen että tämän lapsiprosesseille. [7]

Vuonna 2000 konttiteknologioiden esiaste, jails, lisättiin Unix-pohjaiseen FreeBSD

käyttöjärjestelmään. Tämä osiointiratkaisu mahdollisti prosesseille täydet oikeudet tiedostoihin ja prosesseihin jail-osion sisällä, mutta myös esti pääsyn ja näkyvyyden ulkopuolelle. [8, 9]

Vuonna 2001 konttiteknologiat päätyivät Linuxiin, kun Jacques Gélinas loi VServer-projektin. Projektin version 0.0 muutoslokitiedosto (*change log*) kertoi sen mahdollistavan useiden Linux-palvelimien ajamisen yhdessä koneessa itsenäisesti ja korkealla tietoturvan tasolla [10]. Linux-VServer oli ensimmäinen ratkaisu, jolla pyrittiin erottelemaan ympäristö selkeisiin yksiköihin, virtuaalipalvelimiin. Jokainen virtuaalipalvelin tuntuu ja muistuttaa tavallista palvelinta sen sisältämien prosessien näkökulmasta. Ongelmana oli kuitenkin se, että kyseinen ratkaisu vaati muunnellun version Linuxin ytimestä. [11]

Paul Menagen vuonna 2007 esittämä lähestymistapa sai konttiteknologiat taas liikahtamaan eteenpäin. Hän esitti geneeristen prosessiryhmien (*Generic Process Container*) lisäämistä Linuxin ytimeen [12]. Tämä ratkaisu mahdollisti prosessien yhteen ryhmittelemisen ja takasi jokaiselle ryhmälle muistia, laskenta-aikaa sekä luku- ja kirjoitusaikaa levylle ilman, että yksi ryhmä olisi saanut kaiken käyttöönsä [11].

Yksi avain asemassa olevista ratkaisuista konttiteknologioissa tapahtui, kun nimiavaruudet (*namespace*) tulivat Linuxiin vuonna 2008. Linuxin prosessihierarkia toimi perinteisesti siten, että ydin ylläpitää puurakennetta prosesseista. Nimiavaruudet mahdollistivat prosessin aloittavan oman alipuunsa, jossa aliprosessit eivät enää olleet tietoisia ylemmän tason prosessien olemassa olosta. Nimiavaruudet mahdollistavat myös pääkäyttäjän oikeuksien käytön aliprosesseissa, mutta ei niiden ulkopuolella. [11, 13]

Vuonna 2008 julkaistiin ensimmäinen versio LXC:stä. LXC, eli Linux Containers, on käyttöjärjestelmätason virtualisointiin tarkoitettu projekti. Sen tarkoituksena on luoda säiliöitä, jotka ovat niin lähellä virtuaalikonetta kuin suinkin mahdollista, mutta käyttäen silti samaa kerneliä ja ilman ylimääräisiä ”kustannuksia”, jotka laitteiston emuloinnista syntyy. [14]

Docker julkaistiin avoimen lähdekoodin projektina vuonna 2013 [15]. Docker pohjau-

tuu aiemmin kehitettyihin teknologisiin ratkaisuihin, kuten nimiavaruuksiin, mutta vasta Dockerin myötä konttiteknologioiden suosio räjähti kasvuun etenkin Dockerin helppokäyttöisyyden vuoksi.

3.2 Arkkitehtuuri

Tärkeimmät teknologiset mahdollistajat konteille perustuvat Linux-käyttöjärjestelmän ytimen ominaisuuksiin, joilla mahdollistetaan (1) prosessien ajon looginen erottaminen ja eristäminen sekä (2) resurssien rajoittaminen prosesseille [16]. Konttiteknologioita voidaan ajatella eräänlaisena virtualisoinnin muotona, mutta sen sijaan, että virtuaalisen laitteiston päällä ajettaisiin kokonaista käyttöjärjestelmää, nykypäivän konttiteknologiaan perustuva virtualisointi rakentuu Linux-ytimen nimiavaruuksien päälle. Konttiteknologia konseptina voidaankin nähdä yhtenä hallinnan tasona käyttäjien (*user*) ja ryhmien (*group*) lisäksi, vaikkakin se on toteutukseltaan monimutkaisempi. Nimiavaruuksia voidaan käyttää eri tavoin, mutta yleisin lähestymistapa on luoda eristettyjä tiloja, tai konteja, joilla ei ole mahdollisuutta havaita tai vaikuttaa prosesseihin tai resursseihin itsensä ulkopuolella.

Vaikka konttiteknologioiden avulla pienimmillään yksi kontti voi ajaa vaikkapa yhtä prosessia, on toki mahdollista sisällyttää kokonainen käyttöjärjestelmä konttiin. Konttia, joka pitää sisällään kokonaisen käyttöjärjestelmän ja ajaa prosesseja kuten `init`, `inetd`, `sshd`, `syslogd`, `cron` ym. kutsutaan systeemikontiksi (*system container*). Konttia, joka ajaa vain tiettyä sovellusta, kutsutaan sovelluskontiksi (*application container*). Molemmilla konttityypeillä on omat käyttötarkoituksensa. Koska sovelluskontit eivät tarvitse muistia ylimääräisten systeemiprosessien ajamiseen, kuluttavat ne yleensä vähemmän tietokoneen resursseja. [6]

Resurssien, kuten vaikkapa tiedostojen, jakaminen konttien välillä on helppoa. *bind mount* -tyyppinen kiinnittäminen mahdollistaa yhden kansion jakamisen useiden konttien

välillä ja se voidaan kiinnittää eri konteissa eri lokaatioihin. Tämä on toteutettu tehokkaasti Linuxin VFS-kerroksessa. [6]

3.2.1 Tietoturva

Konttien tietoturvan hallinnalla on taipumus olla helpompaa kuin Unixin käyttäjätasojen hallinta. Tämä johtuu siitä, että kontti ei pääse käsiksi mihinkään, mitä se ei voi nähdä. Näin ollen liian laajojen käyttöoikeuksien tapaukset vähenevät huomattavasti. Kun käytetään hyväksi Linuxin nimiavaruuksia, kontin sisäistä `root`-käyttäjää ei kohdella saman tasoisena kontin ulkopuolella. Ensisijainen tietoturvaongelma säiliöteknologioissa on systeemikutsut, jotka eivät ole tietoisia nimiavaruuksista ja voivat siten aiheuttaa tahattoman vuodon konttien välillä. [6]

Koska konttiteknologioita käytetään useasti mikropalveluarkkitehtuurin mukaisten sovellusten rakentamiseen, toimivat kontit usein yhteydessä toisiinsa erilaisten verkkoon esillä olevien rajapintojen kautta. Tämä mahdollistaa myös luonnollisesti suuremman hyökkäyspinta-alan. Kun useita mikropalveluita kytketään toisiinsa, kasvaa myös verkon kompleksisuus. Lisäksi sovellusten välinen kommunikaatio vaatii tietyn luottamustason konttien välille. Vaikka nämä ongelmat eivät sinäänsä koske konttiteknologiaa itseään, on ne hyvä ottaa huomioon sovelluksia rakennettaessa. [17]

Muita varteenotettavia tietoturvaongelmia on esimerkiksi Docker-isännän luotettavuus, mahdolliset palvelunestohyökkäykset, väärennettyjen pakettien lähettäminen sekä Docker-daemonin mahdollistamat tietoturva-aukot. [18]

3.2.2 Ongelmia

Konttiteknologioissa on vielä useita ratkaisemattomia ongelmia. Osin nämä ongelmat ovat implementaatiokohtaisia, ja ovat siis ratkaistavissa. Eräs ongelmista liittyy siihen, että prosessit kontin sisällä eivät ole tietoisia niille asetetuista resurssirajoitteista. Kontti siis voidaan käynnistää parametreilla, jotka kertovat sen saavan käyttää muistia esimer-

kiksi 512MB, mutta kun käytettävissä olevan muistin määrää tarkastellaan kontin sisältä vaikkapa komennoilla `free` tai `top`, näyttää nämä komennot isäntäkoneella vapaana olevan muistin määrän. [6, 19]

Eräs ongelma koskee konttien sisällä ajettavien prosessien käyttäjiä tilanteissa, joissa konttiin liitetään isäntäkoneelta tiedostoja tai kansioita. Näiden tiedostojen omistajana näkyy kontin sisällä isäntäkoneen käyttäjän ID, jolloin konttia ajettavan käyttäjän ID:n tulee olla sama kuin isäntäkoneella. [20]

Lisäksi löytyy myös useita käyttöjärjestelmä- ja implementaatiokohtaisia ongelmia joita ei vielä ole ratkaistu. Yksi esimerkki tällaisesta on Mac-tietokoneilla esiintyvä ongelma, jossa konttien levynkuvat vievät isäntäkoneelta tilaa, vaikka ne poistettaisiin. [21]

3.2.3 Implementaatiot

Konttiteknologiat itsessään ovat olleet olemassa jo kymmeniä vuosia, mutta niitä soveltavia implementaatioita on tullut markkinoille suurten massojen käyttöön vasta viimeisten vuosien aikana. Yleisimpänä ja tunnetuimpana implementaationa voidaan pitää Dockeria, sen saavuttaman suuren huomion ja käyttäjäkunnan ansioista. Muita sekä Dockerin kanssa saman kaltaisia, että tästä paljonkin poikkeavia ratkaisuja toki löytyy ja kehitetään jatkuvasti. Näitä ovat esimerkiksi Heroku, rkt sekä runC. [22]

Tässä tutkielmassa keskitymme jatkossa konttiteknologioista juuri Dockeriin sen suosion ja laajan käyttäjäkunnan tuomien etujen vuoksi.

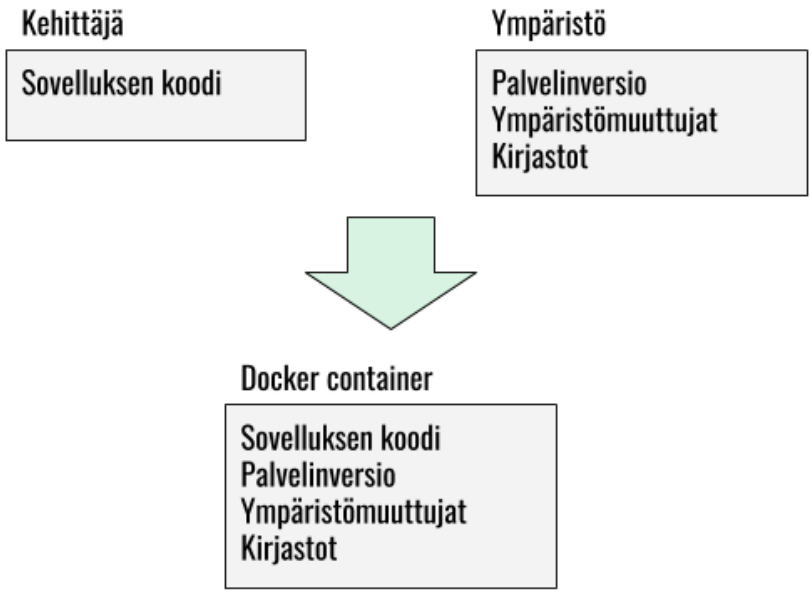
3.3 Docker

3.3.1 Taustaa

Vaikka konttiteknologiat ovat olleet saatavilla Linux-ympäristöihin jo vuosien ajan, ovat ne aiemmin olleet usein vain tiettyyn tarkoitukseen rakennettuja ratkaisuja. Tämä vaikeutti toteutusten uudelleenkäytettävyyttä ja skaalautuvuutta tehden ne joissain tapauk-

sissa jopa mahdottomiksi. Docker on teknologia, joka paikkaa tätä aiemmin olemassa ollutta puutetta esittelemällä standardoidun tavan toteuttaa sekä uudelleenkäytettävyys, eristäminen että skaalautuvuus. Dockerilla onkin kaksi tärkeää roolia: sovellusten paketoinnin formaattina sekä käyttöliittymän ja metodien yhdenmukaistajana.

Docker esiteltiin ensimmäistä kertaa Python kehittäjäkonferenssissa vuonna 2013, kun dotCloud nimisen yrityksen perustaja Solomon Hykes piti lyhyen esittelyn projektista [15, 23]. Projekti sai nopeasti huomattavan määrän mediahuomiota ja se avattiinkin avoimen lähdekoodin projektina verkossa toimivaan versionhallintapalvelu GitHubiin. Nykyään Docker on yksi johtavimmista konttiteknologian sovellusalustoista, jonka avulla kehittäjät voivat helposti eliminoida ”toimii minun ympäristössäni” –tyyppiset ongelmat ja ajaa sovelluksiaan eristetyissä konteissa (*Docker container*). Ideana on se, että sovellus käyttäytyy aina samoin, riippumatta esimerkiksi palvelinympäristöstä tai kehittäjän tietokoneen asetuksista ja asennetuista sovelluskirjastoista. Dockerin voisikin ajatella noudattavan Javan tunnuslausetta: ”Write once, run anywhere”, jonka tarkoituksena on ilmentää kielen yhteensopivuutta eri alustojen kanssa [24]. Käytännössä tämä tarkoittaa sitä, että kehittäjä voi rakentaa sovelluksensa kaikkine riippuvuuksineen, ajaa sitä kehitys-, testiympäristöissään ja tämän jälkeen siirtää täsmälleen saman sovellusnipun tuotantoon. Tällä tavoin myös tuotantoasennus on useimmiten huomattavasti yksinkertaisempaa ja nopeampaa, kun kaikki riippuvuudet sovelluksen ajamiseen on selvitetty jo kehitysvaiheessa. Lisäksi samasta levynkuvasta (*Docker image*) voidaan luoda useita ajettavia instansseja ja näin skaalata sovellusta tarpeen tullessa. Kuvassa 3.1 on demonstroitu, miten Docker-kontti mahdollistaa sekä ajettavan sovelluksen koodin että erinäisten ympäristömuuttujien yhdistämisen toistettavaksi kokonaisuudeksi.



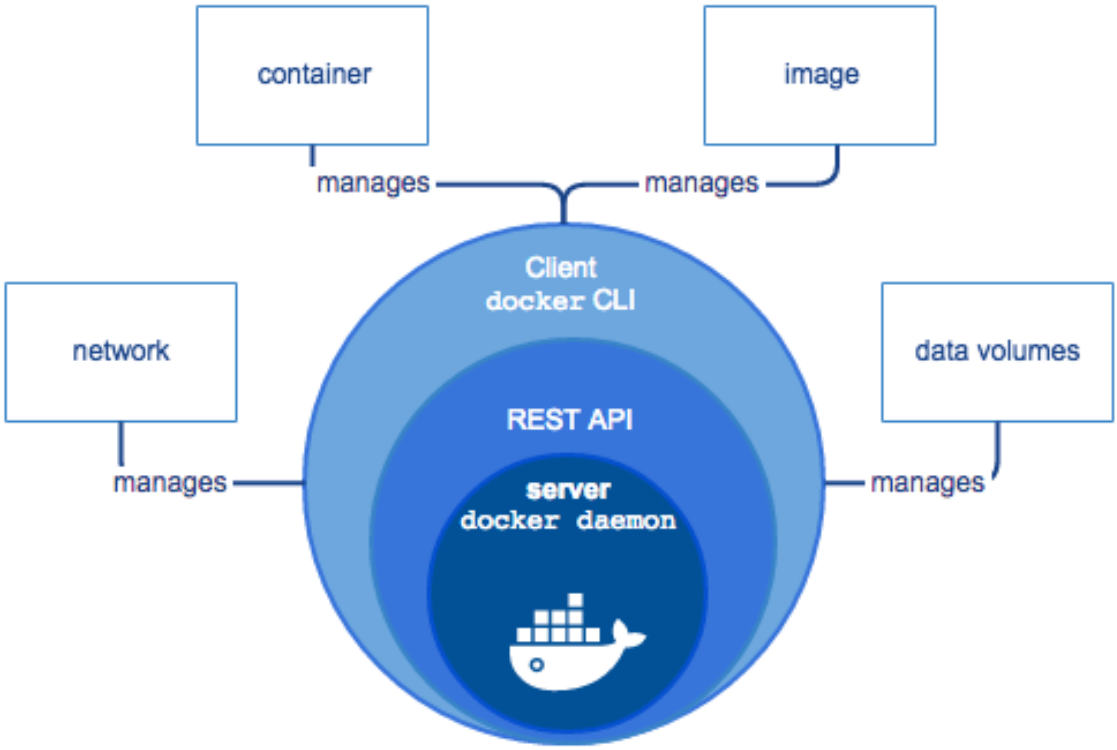
Kuva 3.1: ”Toimii minun ympäristössäni”–ongelma voidaan ratkaista Docker-konteilla

Docker on tuettu suuressa osassa suurimpia pilvipalvelutoimittajia, kuten Amazonin AWS Elastic Beanstalkissa, Google AppEnginessä, IBM Cloudissa ja Microsoft Azuressa. Tämä antaa vahvan signaalin kehittäjille siitä, että vasta joitakin vuosia vanha teknologia tulee pysymään markkinoilla.

3.3.2 Dockerin osat

Vaikka Docker on erittäin voimakas työkalu, ei sen perusrakenne ole kovin monimutkainen. Docker engine on Dockerin ydin ja tarjoaa teknologian, joka mahdollistaa Dockerlevynkuvien (*Docker image*) ja konttien (*Docker container*) luomisen ja ajamisen [25]. Docker engine on asiakas-palvelin sovellus, joka koostuu seuraavista pääkomponenteista: palvelin (*server/docker daemon*), REST-rajapinta (*REST API*) ja asiakassovellus (*docker cli*). Tämä kerrosmainen rakenne käy hyvin ilmi myös Dockerin dokumentaatiosta (kuva

3.2).



Kuva 3.2: Docker enginen rakenne [1]

Asiakassovellus on siis se osa, joka toteuttaa Dockerin REST-rajapinnan, ja jolla käyttäjä antaa komentoja itse palvelimelle. Palvelinsovellusta käytetään komennolla `docker`. Kokonaisuudessaan komento levynkuvien ajamiseen on seuraavanlainen:

```
$ docker run [OPTIONS] IMAGE[:TAG|@DIGEST] \
  [COMMAND] [ARG...]
```

Dockerin komentorivityökalulla voi muun muassa:

- Rakentaa Docker-levynkuvia.

[1] `https://docs.docker.com/engine/docker-overview/`

- Ladata levynkuvia Docker-rekisteristä tai lisätä omia levynkuvia rekisteriin.
- Käynnistää Docker-levynkuvista säiliöitä.
- Käynnistää komentorivikäyttöliittymä säiliöön.
- Luoda tai poistaa Docker-volumeja, sekä Docker-verkkoja

Dockerfile

Jokaisen Docker projektin ydin on Dockerfile. Tämä tiedosto sisältää ohjeistuksen, kuinka rakentaa Docker-levykuva. Dockerfile on siis resepti, joka sisältää tarvittavat tiedostot, ympäristön, muuttujat ja komennot Docker-levykuvan luomiseen. Dockerfilen ensimmäisen määreen tulee olla `FROM`-komento. Tällä komennolla kerrotaan Dockerille, mihin olemassa olevaan Docker-levykuvaan oma uusi levykuva ja lopulta ajettava säiliö pohjautuu. `FROM`-komennon on aina oltava mukana ohjeistuksessa. Mikäli käyttäjä haluaa pienimmän mahdollisen riippuvuuden ja haluaa ajaa esimerkiksi vain binäärimuotoista dataa, voi `FROM`-komennolle antaa erikoismääreen `FROM scratch`, joka tarkoittaa nimensä mukaisesti tyhjästä aloittamista. Dockerfile luodaan käyttämällä melko yksinkertaista syntaksia alla olevan esimerkin mukaisesti:

```
1  FROM ubuntu
2
3  RUN apt-get update && \
4      apg-get install -y nginx
5
6  # Override the default config.
7  ADD ./nginx.conf /etc/nginx/nginx.conf
8
9  WORKDIR /etc/nginx
10
11 # Expose ports.
12 EXPOSE 80
13 EXPOSE 443
14
```

```
15 # Define default command.
16 CMD ["nginx"]
```

Koodiesimerkki 3.1: Esimerkki Dockerfilestä

Yllä olevassa esimerkissä pohjana käytetään Ubuntu Linux-jakelua, johon asennetaan Nginx-palvelinsovellus. Tämän jälkeen sovelluksen oletuskonfuguraatio ylikirjoitetaan omilla, ja asetetaan työkansio. Lopuksi sovellukselle avataan portit 80 sekä 443 ja sovellus käynnistetään.

Dockerfilessä `FROM`-määre on pakollinen ja sen täytyy esiintyä ennen muita mahdollisia määreitä, kuten `ADD`, `COPY`, `ENV`, `EXPOSE`, `LABEL`, `STOPSIGNAL`, `USER`, `VOLUME` tai `WORKIR`. `FROM`-määre voi kuitenkin esiintyä samassa Dockerfile-tiedostossa useaan kertaan, mikäli samalla tiedostolla halutaan luoda useita levynkuvia tai edellistä rakennusvaihetta (*build stage*) halutaan käyttää seuraavan pohjana.

Edellä olevan esimerkin `FROM`-määre voitaisiin kirjoittaa myös seuraavasti: `FROM ubuntu:latest`. Nämä kaksi tapaa kertovat, että käytetään aina Ubuntun viimeisintä saatavilla olevaa versiota, sillä `latest` on myös oletustagi, vaikka sitä ei erikseen ilmoittaisi.

Docker image

Docker-levykuva eli Docker image on tiedostojärjestelmä ja parametrit, joita käytetään ajaessa. Sillä ei koskaan ole omaa tilaa, eikä se ikinä muutu, vaan sitä käytetään itse Docker-säiliöiden luomiseen. Docker-levykuva luodaan komennolla `docker build`.

Docker-levykuva pohjautuu usein johonkin jo olemassa olevaan levynkuvaan, jota käyttäjä voi joko muokata tai lisätä tämän päälle esimerkiksi oman sovelluksensa. Esimerkiksi käyttäjä voi luoda levynkuvan, joka pohjautuu Ubuntu Linux-jakeluun, ja asentaa ja ajaa tämän päällä verkkopalvelinta.

Docker-levykuvat rakentuvat useista kerroksista, joista jokainen vastaa yhtä Dockerfilen määrettä. Esimerkiksi seuraavanlainen Dockerfile koostuu neljästä eri määreestä ja tästä rakennettava levynkuva siis muodostaa neljä kerrosta:

```
1 FROM ubuntu:16.04
2 RUN apt-get update
3 RUN apt-get upgrade
4 RUN apt-get install nginx
5 CMD ["nginx"]
```

Edellä olevan Dockerfilen `FROM`-komento luo kerroksen `ubuntu:16.04` levynkuvasta. Tämän jälkeen loput komennot lisäävät oman kerroksensa tähän siten, että uusi kerros kertoo eroavaisuudet edeltävään. Kun käyttäjä lataa levynkuvan tietokoneelleen esimerkiksi komennolla `docker pull my-nginx-image`, ladataan ja tallennetaan levynkuva kerroksittain. Näin esimerkiksi kaksi levynkuvaa, jotka pohjautuvat samaan levynkuvaan, jakavat keskenään tämän kerroksen [26]. Nämä kerrokset myös tallennetaan välimuistiin, jolloin saman levynkuvan rakentaminen on seuraavalla kerralla nopeampaa. Jotta kerroksia voitaisiin vähentää ja rakennusaikaa tulevilla kerroilla nopeuttaa, edellinen Dockerfile voitaisiin kirjoittaa siis muotoon:

```
1 FROM ubuntu:16.04
2 RUN apt-get update && \
3     apt-get upgrade && \
4     apt-get install nginx
5 CMD ["nginx"]
```

Docker-säiliö

Docker-säiliö tai Docker-kontti on Docker-levykuvasta luotu ajettava instanssi. Se on oletuksena melko hyvin eristetty sekä isäntäkoneesta että muista suoritettavista konteista. Kun säiliö pysäytetään, kaikki siihen ajon aikana tehdyt muutokset katoavat. Näin saa-

vutetaan yksi Dockerin periaatteista siitä, että kontteja voi surutta tuhota, luoda ja monistaa. Docker-kontti voidaan luoda joko Docker-rekisteristä ladattavasta levynkuvasta, tai levynkuvan voi rakentaa oman Dockerfilen avulla. Kontti voidaan myös asettaa ns. taukotilaan (pause), jolloin sen tila pysyy jäädytettynä ja se voidaan käynnistää uudestaan samassa tilassa. Tämä on kätevää etenkin kun kontin sisällä ajettavat prosessit halutaan jäädyttää esimerkiksi tiedostojen varmuuskopioinnin ajaksi. Seuraavassa esimerkissä Docker-konttiin käynnistetään Ubuntu, joka yhdistetään terminaaliin ja siinä ajetaan bash shell:

```
docker run –i –t ubuntu bash
```

Kun tämä komento ajetaan, tapahtuu seuraavaa [27]:

1. Jos sinulla ei ole ubuntu Linux-jakelun Docker-levynkuvaa jo ladattuna tietokoneelle, Docker lataa sen. Sama saavutetaan ajamalla itse komento `docker pull ubuntu`.
2. Docker luo uuden säiliön. Tämä saavutetaan manuaalisesti myös komennolla `docker create`.
3. Docker allokoi tiedostojärjestelmän sekä kirjoitus että lukuoikeuksilla, jota säiliö pystyy käyttämään. Tämä mahdollistaa Docker-säiliölle tiedostojen luomisen ja muokkaamisen sen omassa lokaalissa tiedostojärjestelmässä.
4. Docker luo verkkoympäristön joka mahdollistaa esimerkiksi ip-osoitteen antamisen säiliöille.
5. Docker käynnistää säiliön ja suorittaa bashin. Koska annoimme komenolle parametreina `–i` ja `–t`, käynnistyy säiliö interaktiivisessa tilassa ja yhdistää se bashin käyttäjän terminaaliin. Näin voimme antaa ubuntulle komentoja terminaalin kautta.
6. Kun kirjoitat `exit`-komennon, Docker sulkee bashin ja säiliö pysähtyy, mutta sitä ei poisteta vaan sen voi käynnistää uudestaan.

Docker volume

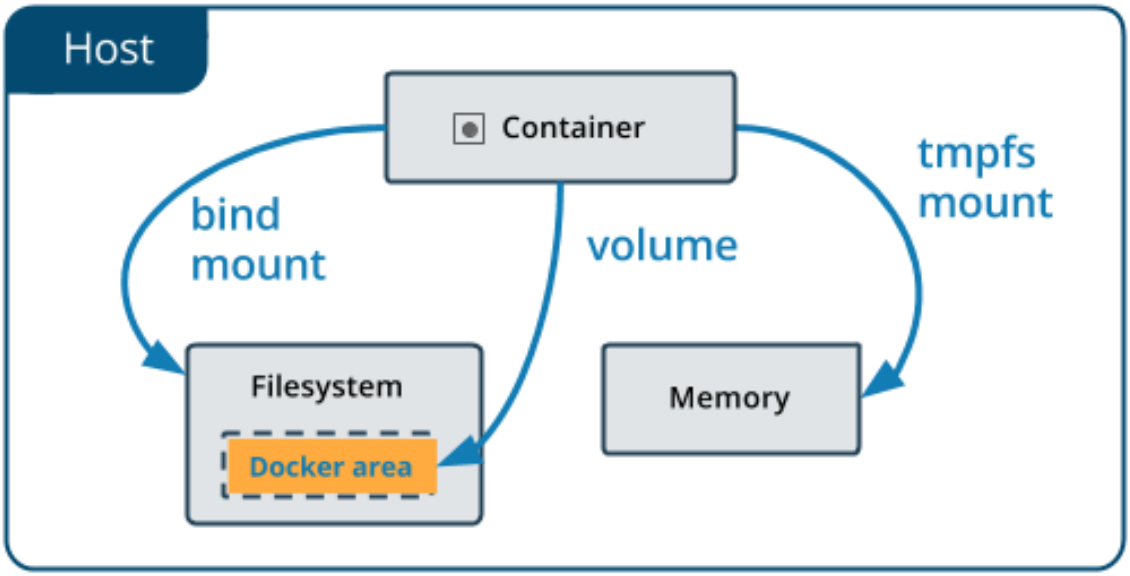
Dockerissa säiliön datan pysyvyyden takaamiseksi on kaksi tapaa: `bind mount` sekä `volume`. Nämä siis mahdollistavat sen, että kun säiliö tuhotaan, sen tallentama data ei häviä. Bind mountilla tarkoitetaan isäntäkoneen kansion tai tiedoston kiinnittämistä containeriin ja se on siis riippuvainen isäntäkoneen kansio- ja tiedostorakenteesta. Docker volumet ovat taas täysin Dockerin hallinnoimia ja niillä on useita etuja verrattaen bind mountiin, kuten:

- Ne ovat helpommin[2] varmuuskopioitavissa sekä siirrettävissä
- Niitä voi hallita Dockerin komentorivityökalujen kanssa
- Ne toimivat samalla tavoin kaikissa käyttöjärjestelmissä
- Ne ovat helpommin jaettavissa useiden konttien kesken
- Ajurit mahdollistavat volumien käytön toiselta koneelta tai pilvipalvelimelta, niiden kryptaamisen sekä muita toiminnallisuuksia

Lisäksi volumet ovat usein parempi vaihtoehto itse konttiin kirjoittamiselle myös siksi, että ne eivät kasvata itse ajettavan kontin kokoa, ja data säilyy itse kontin elinkaaresta huolimatta [28].

Mikäli kontille halutaan antaa pääsy dataan ilman, että sitä kirjoitetaan mihinkään, voidaan käyttää `tmpfs mount` tyyppistä kytkentää. Tämä mahdollistaa datan kirjoittamisen isäntäkoneen muistiin. Kuvassa 3.3 käy hyvin ilmi miten nämä kolme erityyppistä datakytkentää vaikuttavat isäntäkoneeseen.

[2]Operaatio on helpompi suorittaa toisen kontin avulla, sillä tiedostojen sijaintia isäntäkoneella ei tarvitse tietää



Kuva 3.3: Dockerin datakytkentätavat

Docker volumeja käskytetään käyttämällä `docker volume` komentoja. Volume luodaan esimerkiksi seuraavanlaisella komennolla:

```
docker volume create my-vol
```

Muita mahdollisia parametreja `docker volume` -komennolle ovat `inspect`, `ls`, `prune` sekä `rm`. Näiden avulle voidaan volumesta näyttää lisäinformaatiota, listata kaikki volumet, poistaa kaikki käyttämättömät volumet tai poistaa yksi tai useampi volume. Kun volume on luotu, voidaan se liittää käynnistettävään konttiin esimerkiksi seuraavasti:

```
docker run -d -it -v my-vol:/app nginx:latest
```

Komento käynnistää nginx kontin, jonka kansioon /app se sitoo edellisessä esimerkissä luodun volumen. Volume voidaan myös liittää konttiin tilassa, jossa kontilla on ainoastaan lukuoikeudet sen sisältöön käyttämällä `:ro` päätettä:

```
docker run -d -it -v my-vol:/app:ro nginx:latest
```

Docker-verkot

Docker-verkot (*Docker network*) on Dockerin oletustapa kommunikoida konttien välillä. Docker verkkoja komennetaan `docker network` -komennoilla. Docker luo asennuksen yhteydessä kolme oletusverkkoa, joita voi tarkastella `docker network ls` komennolla:

```
docker network ls

NETWORK ID          NAME        DRIVER          SCOPE
b81c4682a6f9        bridge      bridge          local
6c7325ba8f7b        host        host            local
2fcc9b521bc2        none        null            local
```

Näiden verkkojen kautta ajettavat kontit voidaan yhdistää toisiinsa esimerkiksi käyttämällä `--network` parametria käynnistyksen yhteydessä. Näistä oletuksena käytetään aina `bridge`-verkkoa. `none`-verkon käyttö lisää kontin konttikohtaiseen verkkopinoon ja näin kontilta puuttuu verkkoliitäntä. Käyttämällä `host`-verkkoa, liitetään kontti suoraan isäntäkoneen verkkoon. Tämä vaikuttaa myös kontin eristykseen, sillä jos esimerkiksi kontti ajaa verkkopalvelinta portissa 80, on tämä palvelin automaattisesti saatavilla myös isäntäkoneen portissa 80. [29]

Docker machine

Docker machine on työkalu, joka mahdollistaa useiden Docker engine -sovellusten ajamisen samassa tietokoneessa. Tällöin muun muassa useiden eri versioiden ajaminen samalla tietokoneella on mahdollista. Koska Docker engine toimii natiivisti vain Linux-ympäristöissä, voi Docker engineä ajaa siinä suoraan vain asentamalla sen. Docker mac-

hinen avulla komennolla `docker-machine`, käyttäjä voi kuitenkin ylläpitää helposti suurtakin määrää Docker isäntiä. Docker machine luo automaattisesti isännät, asentaa Docker enginet niihin ja konfiguroi Docker asiakassovellukset niihin.

Docker-compose

Mikropalvelu-arkkitehtuurin konsepti on valtaamassa alaa ja monoliittiset sovellusmallit ovat hiljalleen väistymässä näiden tieltä. Ideana on toteuttaa suurempi palvelu joukkona pieniä sovelluksia, jotka hoitavat yhden tarkoin määritellyn tehtävän autonomisesti, mutta toimivat yhteistyössä toisten sovellusten ja palveluiden kanssa. Docker, ja konttiteknologia yleisesti, toteuttaa juuri tämän kaltaista mallia, jossa yhden Docker-kontin tarkoitus on palvella vain tarkoin määriteltyä tehtävää.

Konttien linkittäminen onkin yksi Dockerin merkittävimmistä ominaisuuksista. Kontit voidaan linkittää toisiinsa ja täten tehdä ne tietoisiksi toisistaan. Linkitetyillä konteilla on ns. lähettäjä-vastaanottaja suhde, jossa lähettäjä linkitetään vastaanottajaan, ja vastaanottava voi turvallisesti ottaa vastaan informaatiota ja dataa lähettäjältä. [30]

Mutta miten sitten koostaa kokonaisuus yksittäisistä Docker-konteista? Docker Compose on työkalu tähän tarkoitukseen:

> ”Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a Compose file to configure your application’s services. Then, using a single command, you create and start all the services from your configuration.” [31]

Docker Compose –työkalun avulla voidaan määritellä ja ajaa Docker sovelluksia, jotka koostuvat useista konteista ja nämä voidaan linkittää toisiinsa. Dokumentaatio kertoo, että työkalun käytön yleisimmät vaiheet ovat seuraavat [31]:

1. Määrittele sovelluksesi `Dockerfile` tiedoston avulla siten, että se voidaan toistaa kaikkialla.

2. Määrittele kaikki tarvittavat palvelut sovellukseesi, jotta ne toimivat yhteydessä toisiinsa `docker-compose.yml` konfiguraatiotiedoston avulla.
3. Aja `docker-compose up` komento käynnistääksesi koko sovelluksesi.

Määriteltävä `docker-compose.yml` tiedosto voi näyttää esimerkiksi seuraavanlaiselta:

```
version: '3'
services:
    web:
        image: nginx
        ports:
            - "8080:80"
        links:
            - php
    db:
        image: mariadb
        ports:
            - "3306:3306"
    php:
        image: php-fpm
        links:
        - db
```

Koodiesimerkki 3.2: Esikerkki docker-compose -tiedostosta

Tiedostossa siis määritellään käytettävät Docker-levynkuvat kukin omana palvelunaan. Esimerkkitiedoston mukainen konfiguraatio käynnistää kolme Docker-konttia, ensimmäiseen nginx www-palvelimen, toiseen mariadb-relaatiotietokantapalvelun ja kolmanteen php-fpm PHP-tulkin. Palvelut voidaan konfiguroida linkittymään toisiinsa, jotta ne voivat käyttää toistensa palveluita. Edellä mainitussa esimerkissä PHP voi kirjoittaa tietokantaan ja nginx pystyy ohjaamaan `.php` tiedostot PHP-tulkille. Paitsi että konttien linkittäminen mahdollistaa resurssien jakamisen, se myös määrittelee konttien käynnisty-

misjärjestyksen. Esimerkin mukaisessa asetelmassa `mariadb` käynnistyy ensimmäisenä, seuraavana käynnistyy `php-fpm` ja viimeisenä `nginx`. Linkittäminen siis myös määrittelee kontit, joiden tulee olla käynnissä ennen kuin kyseinen kontti voidaan käynnistää. Compose-tiedoston avulla voi levynkuvien lisäksi antaa käynnistettävälle kontille myös Dockerfilen:

```
1 version: '3'
2 services:
3     web:
4         build:
5             context: ./nginx
6         ports:
7             - "8080:80"
```

Tässä tapauksessa annetaan siis kansio, josta Compose osaa automaattisesti etsiä Dockerfile nimistä tiedostoa. Edeltävässä esimerkkitapauksessa hakemistorakenne olisi siis seuraavanlainen:

```
docker-compose.yml
nginx/
    |
    |--Dockerfile
```

3.3.3 Docker-rekisteri

Docker-rekisterillä (*Docker Registry*) tarkoitetaan tilatonta ja skaalautuvaa palvelinsovellusta, jonka avulla pystytään tallentamaan ja jakelemaan Docker-levynkuvia. Rekisteri on avoimen lähdekoodin sovellus, joka toimii Apache-lisenssin alla [32]. Docker ylläpitää itse omaa rekisteriä osoitteessa `https://hub.docker.com/`, mutta käyttäjät voivat halutessaan ylläpitää myös omia yksityisiä rekistereitä. Rekisteriä voi ajatella Dockerin vastineena GitHubille, eli sielä voi ylläpitää keskitetysti omia levynkuvia ja niiden eri

versioita, sekä automatisoida niiden uudelleenrakentaminen esimerkiksi kun tietyt riippuvuudet päivittyvät.

Dockerilla on myös toinen rekisteri-palvelu, nimeltään Docker Cloud. Docker Cloud on palvelu, joka sisältää rekisterin lisäksi myös toiminnot Docker-levynkuvista luotujen sovellusten rakennus- ja testausympäristönä. Sen avulla käyttäjät pystyvät hallitsemaan ja ylläpitämään Docker-infrastruktuuriaan, sekä automatisoimaan mm. niiden elinkaarta. [33]

3.3.4 Orkesterointi

Kun konttien määrä kasvaa satoihin tai tuhansiin, tarvitaan avuksi niin sanottua orkesterointia. Konttien orkesteroinnilla tarkoitetaan automaattista konttien hallintaa, jonka avulla kontteja voidaan esimerkiksi skaalata automaattisesti ja hallita sekä kontteja että levynkuvia esimerkiksi useista Docker-isännistä koostuvassa verkostossa. Docker-konttien orkesteroinnin keskeisimmät sovellutukset ovat Docker Swarm sekä Kubernetes:

Docker Swarm

Docker Swarm on Dockeria kehittävän Docker Inc:n oma orkesterointityökalu konttien hallintaan ja näin se onkin suoraan integroituna Dockerin omiin työkaluihin. Docker Swarmin avulla voidaan hallita niin konttien klusterointia, skaalausta, palveluiden löytyvyyttä, kuormantasausta kuin päivittämistäkin.

Kubernetes

Kubernetes on yksi suosituimmista orkesteroinnin työkaluista. Se on alunperin Googlen kehittämä ohjelmisto ja pohjautuu yrityksen sisäiseen Borg-nimiseen järjestelmään. Kubernetes tukee Dockerin lisäksi myös muita konttiteknologioita.

3.3.5 Taustalla oleva teknologia

Docker on kirjoitettu Go-ohjelmointikielellä ja se käyttää hyväkseen useita Linux-ytimen ominaisuuksia, kuten nimiavaruuksia sekä kontrolliryhmiä, toimintansa takaamiseksi. [27]

Nimiavaruudet

Docker käyttää teknologiaa nimeltään nimiavaruudet (*namespaces*), joiden avulla konteille saadaan eristyskerros. Jokainen oma nimiavaruus toteuttaa yhden tällaisen kerroksen ja Dockerissa on käytössä mm. seuraavia Linux nimiavaruuksia [27]:

- **`pid` nimiavaruus:** Prosessien eristämiseen (*Process isolation*)
- **`net` nimiavaruus:** Verkkorajapintojen hallintaan (*Network interfaces*)
- **`ipc` nimiavaruus:** Prosessien väliseen kommunikaatioon (*IPC resources*)
- **`mnt` nimiavaruus:** Tiedostojärjestelmien kiinnityksiin (*filesystem mounts*)
- **`uts` nimiavaruus:** Laskenta-aikojen jakamiseeen (*Unix Timesharing System*)

Kontrolliryhmät

Docker Engine käyttää myös teknologiaa nimeltään kontrolliryhmät (*cgroups*). Kontrolliryhmän avulla sovelluksen käytössä olevaa laskentatehoa ja resursseja voidaan rajata. Kontrolliryhmien avulla Docker kontit voivat siis käyttää isäntäkoneen resursseja ja toisaalta konteille voidaan pakottaa tiettyjä resurssirajoituksia. Näin voidaan esimerkiksi rajoittaa yhden kontin käytössä olevan muistin määrää. [27]

Union file systems

Union file systems eli UnionFS, on tiedostojärjestelmäpalvelu. Docker Engine käyttää UnionFS-järjestelmää taatakseen tietojärjestelmätyökalut konteille. Docker Engine voi

käyttää useita UnionFS-variantteja, kuten `AUFS`, `btrfs`, `vfs` sekä `DeviceMapper`. [27]

libcontainer

Docker Engine koostaa nimiavaruudet, kontrolliryhmät sekä UnionFS-tiedostojärjestelmäpalvelut yhdeksi konttiformaatiksi. Tätä formaattia kutsutaan nimellä `libcontainer`. [27]

3.3.6 Dockerin käyttöönotto

Tässä aliluvussa käyn läpi lyhyesti Dockerin käyttöönoton. Docker on saatavilla kahtena erillisenä versiona; Docker Enterprise Edition (Docker EE) sekä Docker Community Edition (Docker CE). Koska EE -versio on tarkoitettu lähinnä suurten yritysten käyttöön, käytän ja viittaan läpi tutkielmani Dockerin CE versioon. Docker CE on saatavilla Linux-päätelaitteiden lisäksi myös Windows- ja macOS-tietokoneille, mutta olen rajannut käyttöönoton Ubuntu linux-distribuutioon, koska sekä Mac- että Windows-tietokoneilla käyttöönotto tapahtuu lähinnä asentamalla yksi sovellus. Asennuksen vaiheet ovat seuraavat [34]:

1. Päivitä `apt` pakettihakemisto:

```
sudo apt-get update
```

2. Asenna vaadittavat riippuvuudet:

```
sudo apt-get install apt-transport-https \
     ca-certificates curl \
     software-properties-common
```

3. Lisää Dockerin virallinen GPG avain:

```
curl -fsSL \
    https://download.docker.com/linux/ubuntu/gpg | \
    sudo apt-key add -
```

4. Lisää Dockerin vakaan version repositorio itsellesi (komento saattaa vaihdella järjestelmäarkkitehtuurin mukaan):

```
sudo add-apt-repository "deb [arch=amd64] \
    https://download.docker.com/linux/ubuntu \
    $(lsb_release -cs) stable"
```

5. Päivitä `apt` pakettihakemisto:

```
sudo apt-get update
```

6. Asenna uusin Docker CE versio:

```
sudo apt-get install docker-ce
```

7. Varmista asennuksen toimivuus ajamalla asennuken mukana tuleva `hello-world` levynkuva:

```
sudo docker run hello-world
```

Dockerin compose-työkalu ei tule mukana Docker CE asennuksessa, vaan se on asennettava erikseen. Sen asentaminen hoituu kuitenkin vain muutamalla lisäkomennolla. Seuraavassa vaiheet composen asentamiseen:

1. Lataa uustin Docker Compose versio (vaihda mahdollisesti versionumero uudempaan):

```
sudo curl -L \
    https://github.com/docker/compose/releases/\
    download/1.18.0/\
    docker-compose-`uname -s`-`uname -m` -o \
    /usr/local/bin/docker-compose
```

2. Aseta tarvittavat suoritusoikeudet sovellukselle:

```
sudo chmod +x /usr/local/bin/docker-compose
```

3. Testaa asennuksen toimivuus kysymällä sovelluksen versiota:

```
docker-compose --version
```

3.3.7 Docker osana ohjelmistotuotantoa

Docker ja konttiteknologiat ovat erittäin toimiva konsepti, kun tarve on ajaa niin sanottuja tilattomia sovelluksia. Tämä tarkoittaa sitä, että sovelluksen tila säilytetään esimerkiksi erillisessä tietokannassa sovelluksesta erillään. Näin monistettavuus sekä konttien luonne uudelleen luotavina instansseina toimii hyvin. Seuraavassa luvussa käyn läpi, miten Docker toimii osana ohjelmistotuotantoa, millaisia erilaisia rooleja sillä on, mitä hyötyjä siitä on ja millaisia etuuksia sillä saavutetaan.

Luku 4

Docker osana verkkosovellusten kehitystä

4.1 Johdanto ja taustaa

Docker on yksi käytetyimmistä ja eniten huomiota saaneista konttiteknologian sovelluksista. Dockerin käyttäjien määrä on huimassa nousussa ja se on noussut muun muassa RightScalen vuoden 2015 kyselytutkimuksen 13%:n käyttäjämäärästä ensin vuodessa 27%:iin ja vuonna 2017 on se käytössä jo 35%:lla tutkimuksiin vastanneista [35, 36, 37]. Kehitys- ja tuotanto-operaatioiden (*DevOps*) parissa toimivien henkilöiden Docker-käyttö oli vielä vuoden 2016 alussa tehdyn tutkimuksen [38] mukaan suurelta osin teknologian harjoittelua sekä sillä testailua ja vain pieni osa käytti Dockeria tuotannossa. Tähän syyksi ilmoitettiin teknologiaa jo käyttävien osalta muun muassa tietoturvaongelmat, teknologian epäkypsyys sekä käyttäjien kokemuksen puute [38]. Dockerin oman kyselytutkimuksen mukaan kuitenkin yli 50% vastanneista käyttää Dockeria myös tuotannossa [39].

Suuri osa Dockeria koskevista tutkimuksista keskittyy konttiteknologioihin ja niiden toteutukseen teknisellä tasolla tai vertailee teknologioita joko virtuaalikoneisiin tai muihin tekniikoihin. Dockerin käytöstä sovelluskehityksessä ja sovelluskehityksen tukena on

kuitenkin vielä verrattain vähän tutkimustietoa [40]. Dockerilla kuitenkin toteutetaan ja sitä käytetään pääosin verkkopohjaisiin sovelluksiin. Dockerin vuonna 2016 tekemän kyselytutkimuksen mukaan jopa 78% käyttäjistä käyttää Dockeria verkkosovellusten, kuten verkkosivujen, luomiseen [39].

Tässä luvussa tarkoituksenani on selvittää miten Docker soveltuu sovelluskehityksen eri osa-alueisiin. Pyrin selvittämään teknologian käyttöä verkkosivustojen ja -ohjelmistojen kehityksen osana ja tukena. Käyn kuitenkin läpi myös joitakin esimerkkejä, joissa ei ole kyse verkkopohjaisista sovelluksista, tai niiden käyttö ei välttämättä ainakaan ole rajattu vain verkon kautta käytettäväksi. Käytän tukenani tutkimuksia konttiteknologioiden käytöstä ohjelmistoprosesseissa yleisesti, sekä tutkimuksia niiden käytöstä mm. pilvipalveluiden ja mikropalveluarkkitehtuurien toteutuksissa ja käytössä.

Tarkastelen aihetta niin toisessa luvussa käsittelemieni sovelluskehityksen vaiheiden kautta, mutta myös Dockerin roolia sovelluskehityksessä yleisesti. Mihin Dockeria siis käytetään ja mitkä asiat ovat johtaneet sen räjähdysmäiseen kasvuun? Lopuksi pohdin millaisia vaikutuksia Docker on tuonut nykyaikaiseen sovelluskehitykseen ja millaisia mahdollisuuksia se on avannut sekä mitkä mahdolliset seikat tai ongelmat vielä jarruttavat Dockerin käyttöä?

4.2 Dockerin erilaiset roolit

4.2.1 Mikropalveluarkkitehtuuri

Mikropalveluilla tarkoitetaan arkkitehtuurityyliä, missä ideana on toteuttaa sovelluksia, jotka koostuvat pienistä ja itsenäisistä osista. Nämä osat, eli mikropalvelut, ovat sovelluksen käytettävissä yleensä verkon kautta. Jokainen tällainen osa toteuttaa oman tarkoin määritellyn sovelluksen osakokonaisuuden. Mikropalvelussa siis sovelluksen komponentit, jotka käsittävät sovelluksen, erotetaan toisistaan. [41]

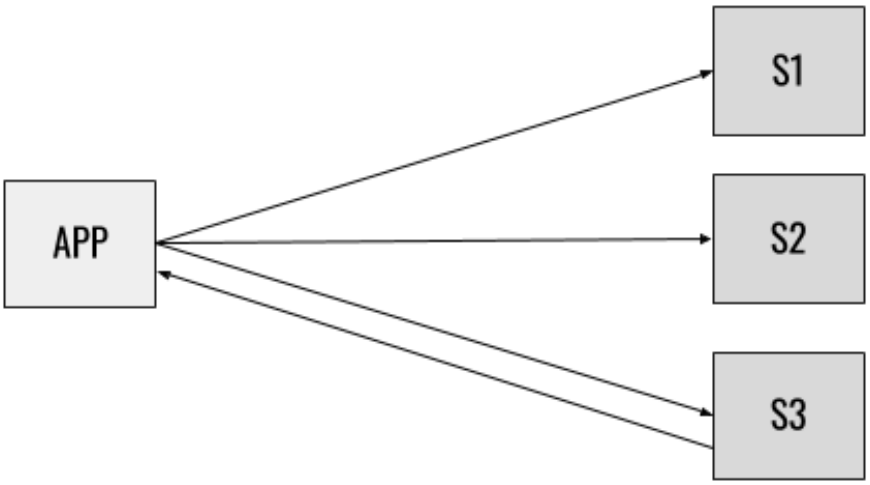
Mikropalveluarkkitehtuurin mukaiset sovellukset voidaan suorittaa itsenäisesti ja niitä

hallitaan usein keskitetysti omasta erillisestä palvelusta. Ne voidaan kukin kirjoittaa eri ohjelmointikielillä ja ne voivat toteuttaa ja käyttää omaa spesifiä datamallia. [42]

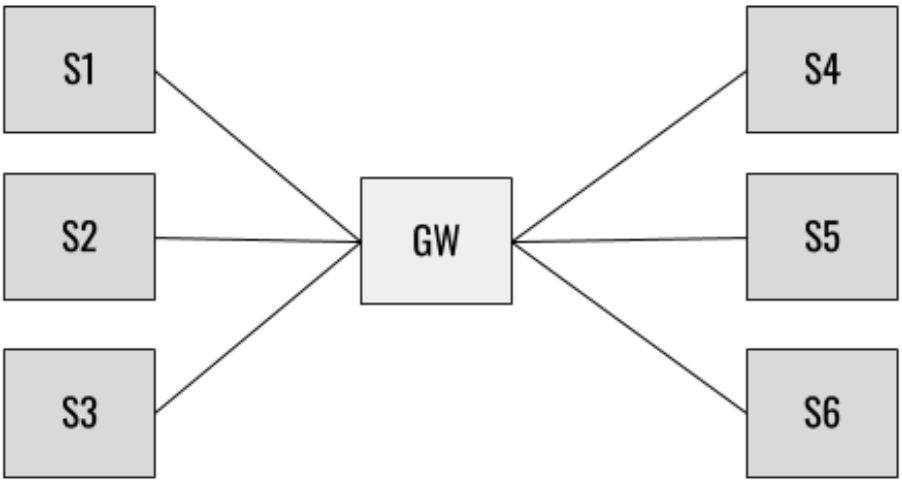
Tämän kaltainen arkkitehtuuri tai metodologia on vastakohta niin sanotuille monoliittisille sovelluksille. Monoliittisella sovelluksella tarkoitetaan sovellusta, joka koostuu suuresta yksittäisestä koodirepositoriosta ja joka yksin tarjoaa kaikki palvelut, kuten HTML-sivut, verkkopalvelut ja rajapinnat yhden sovelluksen sisältä. [43]

Monoliittiset sovellukset vaikeuttavat usein kehittäjien työskentelyä itsenäisesti, sillä ne vaativat paljon koordinaatiota kehittäjien kesken. Myös jatkuva toimitus (luku 4.2.3) vaikeutuu, sillä pientenkin komponenttien päivittäminen vaatii lähes aina koko sovelluksen uudelleen asentamista. Lisäksi yksittäisten komponenttien skaalaaminen monoliittisessa mallissa ei ole mahdollista, vaan skaalaus tapahtuu tällöin aina koko sovelluksen tasolla - ja tällöinkin se on mahdollista vain yksiulotteisesti. [42]

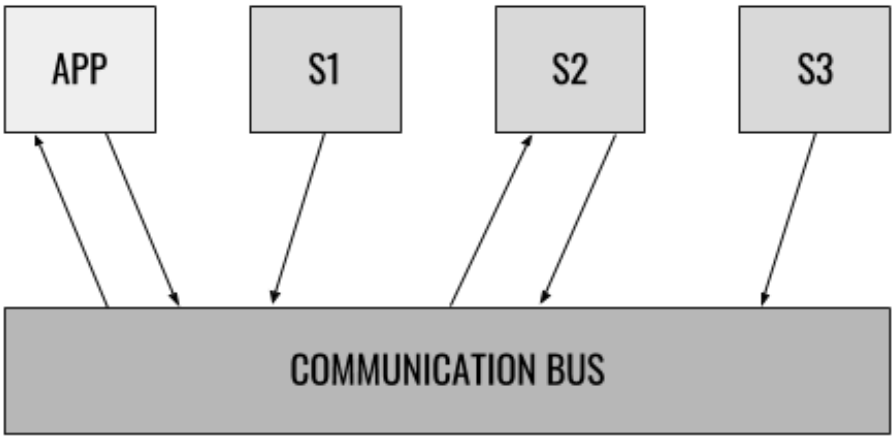
Koska mikropalveluarkkitehtuurissa mikrosovellukset toimivat itsenäisesti, tarvitsevat ne tavan kommunikoida keskenään. Ensimmäinen ja ilmeisin tapa on se, että sovellukset ovat suorassa yhteydessä toisiinsa (kuva 4.1). Se onkin usein kaikkein yksinkertaisin tapa toteuttaa kommunikaatio sovellusten välillä. Suurin potentiaalinen ongelma on mahdolliset viiveet kun kutsujen määrä kasvaa tarpeeksi. Toinen tapa onkin toteuttaa portti (*gateway*), jonka kautta kutsut kulkevat (kuva 4.2). Portin kautta tapahtuvat kutsut voidaan tarvittaessa tallentaa välimuistiin. Portti voidaan tietysti toteuttaa myös omana mikropalvelunaan. Kolmas vaihtoehto on toteuttaa viestintäväylä (*bus, communication bus*), jossa viestit kulkevat asynkronisesti (kuva 4.3). Palvelut voivat rekisteröityä viestintäväylään ja tämän jälkeen käyttää sitä viestien välittämiseen. Tätä tapaa kutsutaan myös julkaise-tilaa -malliksi (*publish-subscribe pattern*). [42]



Kuva 4.1: Suorat kutsut palveluiden välillä



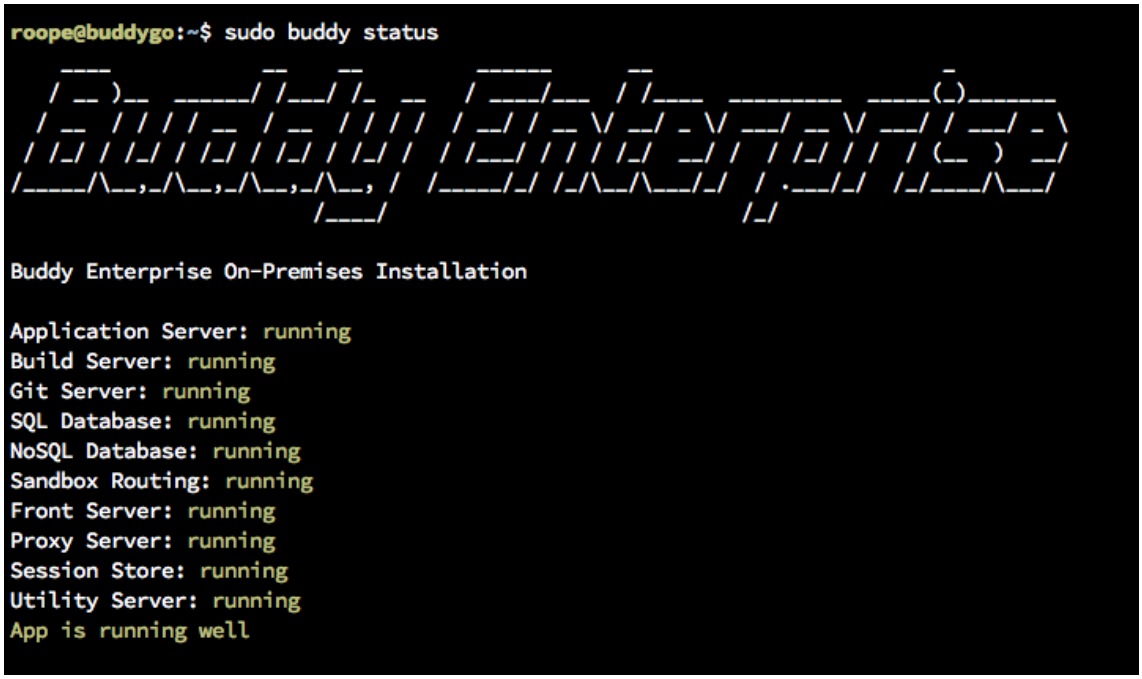
Kuva 4.2: Portti, joka kautta palveluiden viestintä kulkee



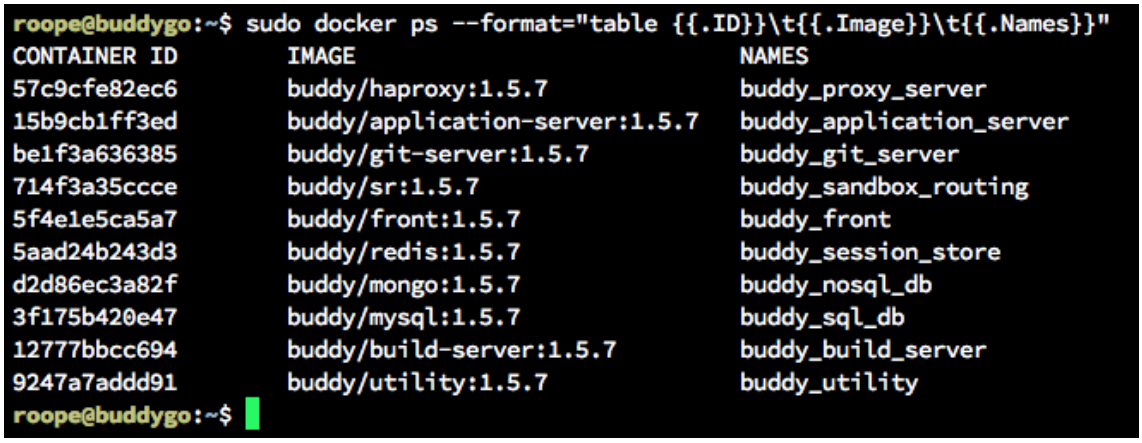
Kuva 4.3: Asynkroniseen viestintään soveltuu julkaisu-tilaus mallin mukainen viestintäväylä

Mikropalveluihin keskittyvästä sovelluskehityksestä saadaan se hyöty, että sovelluksen osia voidaan kehittää erillään toisistaan. Docker on tähän erinomainen valinta, koska sen avulla ohjelman osat voidaan helposti paketoida omiksi kokonaisuuksikseen. Docker myös helpottaa sovellusten suunnittelua siten, että ne suunnitellaan jo alun alkaen toimimaan itsenäisinä piensovelluksina, koska sen toimintaperiaate tukee mallia. Mikropalveluarkkitehtuurin toteuttaminen nähdäänkin yhtenä Dockerin tärkeimmistä eduista [39].

Dockerin avulla esimerkiksi verkkosovelluksen käyttöliittymän osia voidaan rakentaa toimimaan omina mikrosovelluksinaan ja nämä voidaan liittää yhteen Docker-verkon (tai -verkkojen) kautta. Tai verkkosovellus voidaan koota eri tarkoituksiin tarkoitettujen taustasovellusten yhdistelmästä, kuten esimerkiksi Buddy:n (`https://buddy.works`) Enterprise On-Premises jatkuvan integraation sovellus toimii (Kuvat 4.4 ja 4.5).



Kuva 4.4: Buddy Enterprise On-Premises toimii mikropalveluiden kokonaisuutena



Kuva 4.5: Buddyn mikropalveluita ajetaan Docker-konteissa

Yksi syy mikropalveluarkkitehtuurin käytön implementointiin liittyy sen mahdollisuuksiin ajaa sovelluksia pilvipalveluissa. Mikropalveluarkkitehtuuri mahdollistaa suurten koodikirjastojen ja -kokonaisuuksien helpomman hallinnan ja palvelut voidaan toteuttaa kullekin sovellukselle itselleen sopivimmalla ohjelmointikielellä. Mikropalvelumallin avulla ohjelmistokehityksestä saadaan näin ketterämpää ja skaalautuvampaa, ja sovelluk-

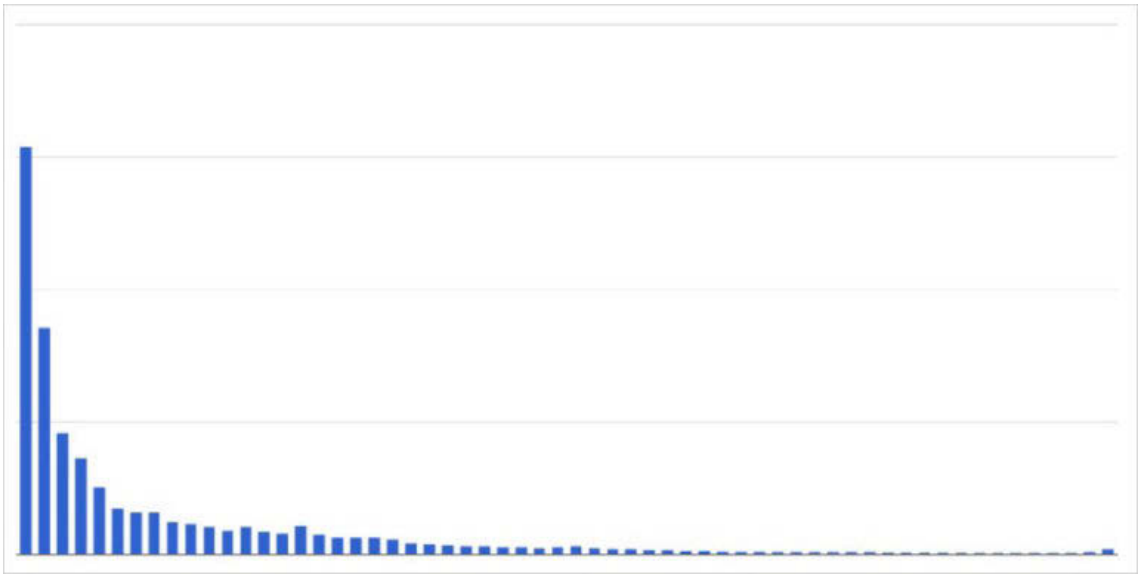
set voidaan ajaa vaikkapa hajautetusti jokainen omassa pilvipalvelussaan. [43]

4.2.2 Pilvipalvelut ja skaalautuvuus

Pilvipalvelut, pilvilaskenta ja skaalautuvuus ovat keskeisiä käsitteitä kun puhutaan konteista ja Dockerista. Dockerin yksi tärkeimmistä periaatteista on konttien uudelleenkäytettävyys ja niiden yksinkertaisuus siinä mielessä, että eriytetään Docker-kontti suorittamaan vain spesifiä ongelmaa.

Google kertoo ajavansa kaikkia palveluitaan Gmailista ja YouTubesta aina Google hakuihin konteissa. Google käynnistääkin viikottain yli kaksi miljardia konttia datakeskuksissaan ympäri maailmaa. Googlen mukaan kontit ovat mahdollistaneet sille vakaamman ja luotettavamman palvelutason sekä korkeamman ja tehokkaamman skaalautuvuuden. [44] Alkuvuonna 2017 myös Netflix ilmoitti omasta merkkipaalustaan skaalautumisessaan ja kertoi käynnistävänsä yli miljoona konttia viikoittain [45].

Nämä luvut kertovat paljon konttien keskimääräisestä eliniästä. New Relic monitorointityökalun kautta kerätystä datasta tehty tutkimus kertoo, että suurin osa sen tietojen mukaan käynnistettävistä konteista on käynnissä vain joitakin minuutteja (Kuva 4.6). [46]



Kuva 4.6: Konttien elinikä minuuteissa tunnin sisällä

Edellä mainittuja seikkoja selittää pitkälti se, että kontteja pystytään käyttämään ns. palvelualustan ulkoistuspalveluissa (*PaaS, Platform-as-a-Service*) minuutti- tai jopa sekuntihinnoittelulla. Näin kontteja voidaan käynnistää kuorman kasvaessa ja sulkea taas heti sen laskiessa. Trendinä onkin käynnistää kontti yhtä HTTP-pyyntöä tai funktionkutsua kohden, suorittaa haluttu tehtävä ja tämän jälkeen kontin voi tuhota. Tämä on erittäin kustannustehokas tapa skaalata palvelua ilman muutoksia palvelutasoon, eikä se vaadi kuorman kasvun ennustamista, sillä tämä on automatisoitu palveluntarjoajan toimesta. Tämän kaltaisella mikropalveluiden ja pilvilaskennan yhdistämisellä voidaankin saavuttaa huomattavia kustannussäästöjä [43].

Kun toteutetaan perinteisempiä verkkosivustoja ja –sovelluksia, ei uudelleenkäytettävyys yksittäisen verkkosivuston kohdalla ole enää niin merkittävä tekijä, mutta etenkin liikennemäärien kasvaessa skaalautuvuus tulee elintärkeäksi osaksi ylläpitoa (luku 2.6). Nykyään monet pilvipalveluiden tarjoajat tarjoavatkin esimerkiksi integroituja orkesterointipalveluita (luku 3.3.4) ja muita hallintapalveluita, joiden avulla sivustoa pystytään skaalaamaan annettujen raja-arvojen puitteissa. Docker levynkuvat ovat erinomainen tapa sivuston ylläpitonäkökulmasta, sillä ne ovat resilienttejä uudelleenkäynnistämiselle ja ne voidaan vaivattomasti konfiguroida skaalattaviksi.

4.2.3 Jatkuva toimitus ja -integraatio

Jatkuva integraatio on ohjelmistokehityksen käytäntö, joka on ollut osana kehitystyötä jo pitkään. Ideana on että kaikki kehitystiimin jäsenet integroivat omat työnsä keskenään, usein useita kertoja työpäivän aikana, mutta vähintään kerran päivässä. Jokainen tällainen integraatio verifioidaan ajamalla sille automatisoitu rakennus (*build*) sekä testit. Näin integroinnin aiheuttamat virhetilanteet havaitaan niin pian kuin mahdollista. [47]

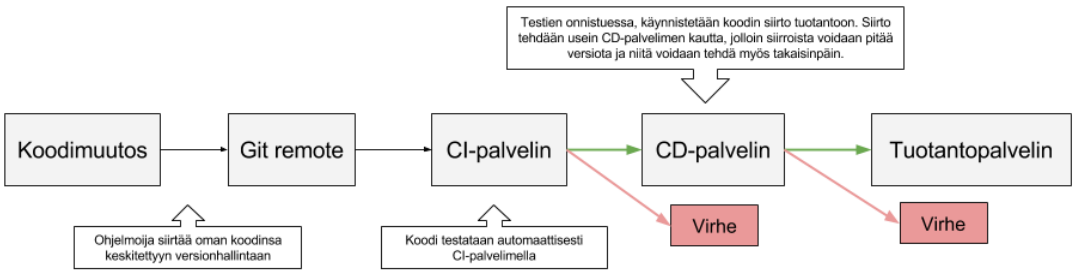
Ensimmäinen vaatimus jatkuvalle toimitukselle ja -integraatioille on keskitetyn versiohallinnan käyttö. Git on yksi yleisimmin käytetyistä versiohallinnan työkaluista. Kun tiimin jäsenet integroivat oman koodinsa keskitettyyn versiohallintaan, tämä koodi ajetaan

ja testataan rakennuksen aikaisten ja ajon aikaisten virheiden osalta. [47]

Jotta tämä kaikki toimisi helposti, on rakentaminen ja testaus oltava automatisoitua. Tätä tehtävää varten löytyy useita palveluita ja työkaluja, kuten vaikkapa Travis CI tai Circle CI.

Docker tekee sovelluksen testaamisen (luku 2.4) tekemisen helpoksi, sillä konttien avulla haluttava ohjelmakoodi ja sen testit voidaan ajaa vaikkapa ohjelmointikielen eri versioilla tai muilla muuttujilla eristetyssä tilassa. Näin suoritettava ohjelma ei pääse vaikuttamaan itse testejä ajavaan isäntäkoneeseen, vaan testit ajetaan turvallisesti kontin sisällä.

Jatkuvalla toimituksella taas tarkoitetaan vielä yhtä lisäpalikkaa prosessiin. Kun ohjelmakoodi on ajettu ja testattu, voidaan se onnistuneen tuloksen jälkeen automaattisesti asentaa. Tämä on tärkeä ominaisuus niin sovelluksen tai verkkosivuston rakennusvaiheessa mutta etenkin kun ollaan siirrytty ylläpitovaiheeseen (luku 2.6), jolloin ajettava koodi on tuotannossa. Tällöin koko ketjun on hyvä toimia automaattisesti kun sovellukseen esimerkiksi tuodaan bugikorjausta (kuva 4.7).



Kuva 4.7: Työvaiheet esimerkiksi bugikorjauksesta tuotantoon CI/CD käytäntöjen avulla

4.2.4 Docker kokonaisvaltaisena ohjelmistokehitysalustana

Dockeria on mahdollista toki käyttää myös kokonaisvaltaisena ohjelmistonkehitysalustana. Tällä tarkoitan siis sitä, että aivan kaikki kehitykseen liittyvä editorista lähtien voidaan ajaa konttien avulla. DockerHubista löytyy esimerkiksi Sublime Text, Microsoftin Visual

Studio Code sekä GitHubin Atom koodieditoreihin valmiit levynkuvat. Ja puuttuvista sovelluksistahan tällaiset voi lähes aina toki tehdä itse.

Mitä hyötyjä tästä sitten saattaisi saada? Otetaan esimerkkinä tilanne, jossa sovelluskehitystiimiin tulee uusi työntekijä. Sen sijaan, että hän joutuisi käyttämään mahdollisesti jopa päiviä asentaakseen kaikki oikeat kirjastot ja riippuvuudet, sekä käytettävät sovellukset ja niiden lisäosat sekä käymään läpi näiden konfiguraatiot, voi henkilö asentaa tietokoneelleen Dockerin, Docker Composen ja käynnistää kehitysympäristön muutamalla komennolla. Kun aivan kaikkea ajetaan konttien sisällä, on jokaisella melko suurella varmuudella tismalleen samat versiot myös esimerkiksi sellaisista kirjastoista ja lisäosista, jotka eivät sinänsä ole osa rakennettavaa ohjelmistoa vaan toimivat apuohjelmina sovelluksen kehittämisen tukena. Tästä esimerkkinä vaikkapa NodeJS, jota ei välttämättä tarvita itse lopullisessa sovelluksessa, mutta sitä käytetään yhdistelemään tarvittavat JavaScript-moduulit. Matt Manning kuvaa hyvin edellisen esimerkin kaltaisen tilanteen kuvitteellisessa tapahtumasarjassa kirjoituksessaan A Tale of Two Onboardings [48]. Näin voidaan myös jäljitellä esimerkiksi toteutettavan verkkosivuston tulevaa palvelinympäristöä, etenkin mikäli sovellusta ei jostain syystä voida tuotannossa ajaa kontissa. Näin saattaa esimerkiksi olla, mikäli käytössä on jaettu palvelintila (luku 2.6.1).

Ongelmina tämän kaltaiseen työskentelyyn saattaa tuoda esimerkiksi aloituskokoonpanon turhan raskas kasaaminen. Tämä on tietenkin suhteessa projektin kokoon. Mikäli projekti on kooltaan pieni, voi yhtä hyvin käydä niin, että aloituskokoonpanon kasaamiseen käytettävä aika ei ole mitenkään järkevässä suhteessa itse projektin kokonaiskestoon. Myös sovellusten päivittäminen voi johtaa ongelmiin kun päivitysrutiinia ei hoideta kaan enää järjestelmän päivitysten yhteydessä. Lisäksi uusien projektien alkaessa saattaa tietokoneelle jäädä kummittelemaan vanhoja levynkuvia vanhoista sovellusversioista tai välimuistista käynnistetään aina päivittämätön levynkuva. Tämä uhkakuva on kuitenkin melko turha, sillä toki uusien työvälineiden tulisi heijastua myös työtapoihin.

4.2.5 Docker työkaluna

Dockerilla työkaluna tarkoitan Dockerin rajoiteuttua käyttöä edellisen kappaleen kokonaisvaltaiseen ohjelmistokehitysalustaan verrattaen. Tällöin siis Dockeria käytetään esimerkiksi vain sovellusten paketoimiseen ja vähentämään ns. ”turhien riippuvuuksien” asentamista työkoneelle. Näin siis pyrkimyksenä on vähentää suoraan käyttöjärjestelmän yhteyteen asennettavia kirjastoja ja sovelluksia. Tällä tavoin esimerkiksi sovelluksista, kirjastoista ja muista riippuvuuksista on helppo ajaa useita eri versioita eri tarkoituksia varten. Mikäli kehittäjä joutuu palaamaan esimerkiksi vuoden takaiseen projektiin, ovat useimmat riippuvuudet saattaneet tässä ajassa jo päivittyä useaan otteeseen. Kun sovellus käyttääkin Docker-kontteja, voi kehittäjä helposti käynnistää sovelluksen sen kehityksen aikaisella versioilla ilman, että omalle koneelle tarvitsee asentaa vanhentuneita paketteja. Näin konttiteknologiaa tuodaan mukaan kehitysvaiheeseen (luku 2.3) ja siitä saadaan verkkosivuston tai -sovelluksen kannalta suuri hyötysuhde. Tästä syystä tämänkaltainen lähestymistapa toimi pohjana myös tapaustutkimukselleni (luku 5).

4.3 Vaikutukset sovelluskehitykseen

Dockeria voidaan käyttää ja soveltaa hyvin laaja-alaisesti verkkosivustojen ja –sovellusten kehityksen eri työvaiheissa (luku 2). Sitä voidaan käyttää apuna etenkin teknisissä työvaiheissa, kuten kehittämisessä, testaamisessa sekä ylläpidossa, mutta Dockerin käyttäminen toki vaikuttaa myös muihin työvaiheisiin. Dockerin tuominen osaksi toteutusta saattaa vaikuttaa esimerkiksi sovelluksen määrittelyyn ja suunnitteluun merkittävästi. Siksi käytettävien teknologioiden ja metodologioiden valintaa on hyvä pohtia jo varhaisessa vaiheessa. Usein suunnan muutokset myöhäisemmässä vaiheessa tulevat kehityksen kannalta kalliiksi ja mikäli vasta myöhäisemmässä vaiheessa halutaan esimerkiksi siirtyä moniliittisen sovelluksen kehittämismallia mikropalveluarkkitehtuuriin, joudutaan työtä tekemään uudestaan alkaen aina määrittelyvaiheesta. Seuraavassa käyn läpi

Dockerin vaikutuksia verkkosovellusten toteutuksen eri työvaiheisiin, jotka ovat esitelty luvussa 2.

4.3.1 Määrittely ja suunnittelu

Vaikka Docker tai muut teknologiavalinnatkaan eivät suoranaisesti vaikuta määrittely- ja suunnitteluvaiheiden suorittamiseen, on niillä merkittävä rooli näiden vaiheiden sisällön kannalta. Kun sovellusta määriteltäessä joudutaan usein lukitsemaan esimerkiksi käytettävät teknologiat (luku 2.1), voidaan Dockerin valinnalla osittain jättää muut teknologiavalinnat myöhempiin vaiheisiin. Näin esimerkiksi käytettävä teknologia ei aseta samanlaisia rajoitteita suunnittelulle kuin se perinteisesti saattaisi tehdä (luku 2.2).

4.3.2 Kehitystyö

Dockerin suurimmat vaikutukset verkkosivustojen toteutuksen työvaiheista koskevat erityisesti sovelluskehitystä (luku 2.3) sekä sovelluksen testaamista, julkaisua ja ylläpitoa (luvut 2.4, 2.5 ja 2.6).

Docker helpottaa uusien tekniikoiden, ohjelmointikielien sekä kirjastojen ja sovelluskehysten opiskelua ja testaamista, sillä konttiteknologia mahdollistaa ns. turvallisen ”hiekkalaatikkoleikin”. Tällä tarkoitetaan sitä, että sovelluksia voi surutta asentaa ja ajaa konteissa ilman vaaraa, että ne vaikuttaisivat isäntäkoneeseen. Asentaminen on myös usein huomattavasti helpompaa, koska Docker-levynkuva sisältää kaiken tarvitsemansa käynnistyäkseen ja toimiakseen. Tämä johtaa myös siihen, että kehittäminen lokaalisti on ns. perinteistä tapaa helpompaa, sillä työympäristön pitäisi ainakin teoriassa olla jokaisella kehittäjällä tismalleen samanlainen esimerkiksi kehittäjän isäntäkoneen käyttöjärjestelmästä riippumatta. Näistä syistä sovelluksen tai sen eri osien teknologiavalintoja on nopea ja helppo testata ja valinta käytettävästä teknologiasta voidaan tehdä vasta tässä vaiheessa. Mikropalveluarkkitehtuurimalli (luku 4.2.1) mahdollistaa myös sovellusten osien toteuttamisen kokonaan eri kielillä tai teknologioilla.

Docker mahdollistaa kehittäjälle täydellisen kontrollin riippuvuuksista, niiden versioista sekä sovelluspäivityksistä. Kun kehittäjä joutuu palaamaan esimerkiksi projektiin, jota kehitettäessä vaikkapa `Node.js` on ollut versiossa 8, ei kehittäjä joudu palauttamaan päivittämäänsä `Node`a versioissa alaspäin omalta koneeltaan taatakseen sovelluksen toimivuuden, vaan versiot voidaan Dockerin avulla lukita projektikohtaisesti. Tämä on oleellista etenkin kun kehitetään useita verkkosivustoja vaikkapa vuoden aikana, sillä käytettävät riippuvuudet päivittyvät tiheään tahtiin. Lukitut ja eristetyt riippuvuudet varmistavat sen, että korjausten ja muutosten tekeminen vanhoihin projekteihin on vaivatonta, eikä kehittäjän tarvitse pelätä lähteekö vanhempi projekti enää käyntiin liian uusilla riippuvuuksilla.

Tietoturvamielessä Docker mahdollistaa kehittäjille paremman kontrollin sovellustensa toiminnasta. Kehittäjä pystyy helposti tiedostamaan ja määrittelemään esimerkiksi mitkä portit mistäkin kontista avataan ulkomaailmaan. Myös jo aiemmin mainittu kontrolli päivityksistä, riippuvuuksista ja asetuksista mahdollistaa sovellukselle vakaamman toimintaympäristön. Tämä toki usein tarkoittaa sitä, että kehittäjän on tiedettävä hieman palvelinympäristön toiminnasta. Docker mahdollistaakin työympäristöissä hieman uudenlaisen roolijaon kun ns. SysAdmin –roolin töitä siirtyy suoraan kehittäjille.

4.3.3 Testaus

Docker-kontit ovat oivallinen väline sovellusten testaamiseen (luku 2.4). Vaikka Docker-kontit eivät välttämättä tuo juurikaan muutoksia verkkosivujen ja -sovellusten selaintestaukseen, etenkin yksikkötestejä ajettaessa kontit ovat erinomainen ratkaisu, koska testit voidaan helposti ajaa esimerkiksi usealla eri ympäristöversiolla. Testejä ajaessa on myös tärkeää, että ulkoiset tekijät pääsevät vaikuttamaan niiden suoritukseen mahdollisimman vähän, joten kontit ovat tähän oiva ratkaisu. Esimerkkitapaus PHP-kielellä toteutetun verkkosovelluksen yksikkötestaamisesta Docker-konttien avulla voisi olla seuraavanlainen:

1. Kirjoita sovellukselle yksikkötestit
2. Paketoi sovellus ja testit Docker-konttiin
3. Aja testit PHP:n versiolla 5.X
4. Aja testit PHP:n versiolla 7.0
5. Aja testit PHP:n versiolla 7.1

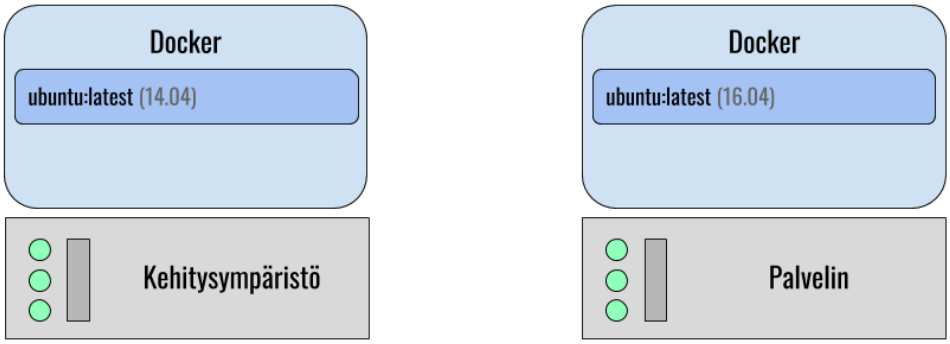
Tätä tapausta varten voitaisiin kirjoittaa Docker-kontti, joka sisältää testikirjastot ja ajettavan PHP:n, jonka asennetta versio voidaan erikseen määritellä. Konttia käynnistäessä tälle voidaan antaa volumena suoritettava sovellus testeineen. Kontti ajaa testit, kertoo kehittäjälle testien tulokset ja sammuu.

4.3.4 Ylläpito

Docker tuo useita uusia ylläpitomahdollisuuksia kehittäjien ulottuville (luku 2.6). Useat suuret pilvipalveluiden tarjoajat, kuten Google tai Amazon, ovat rajanneet tarkasti, minkälaisia sovelluksia heidän palvelimillaan voidaan ajaa. Docker on kuitenkin alustana saatavilla ainakin kaikilla suurimmilla palveluntarjoajilla, ja sovellusten ”paketointi” mahdollistaa sellaistenkin sovellusten ajamisen, joihin pilvipalveluiden tarjoajat eivät muuten tarjoaisi tukea esimerkiksi ohjelmointikielen puolesta.

Dockerin, kuten minkä tahansa uuden teknologian tai työtavan käyttöönottaminen tuo toki mukanaan myös haasteita. Se haastaa kehittäjiä muokkaamaan omaa työtapaansa ja toisaalta koko tekemisen prosessia. Esimerkiksi sovellusten tuottamien lokitiedostojen sijoittamista ja lukemista Docker-konteista täytyy miettiä uudestaan Docker-konttien luonteen takia. Koska kontit ovat siirrettäviä, on yleinen tapa, että ne kirjoittavat oman lokinsa suoraan syötevirtoihin (*stdout, stderr*) ja lokit täytyy lukea ja tätä kautta. Näin kehittäjien täytyy lukea lokit näistä syötevirroista ja ohjata ne mahdollisesti omiin kanaviinsa. Konteja ylläpidettäessä onkin hyvä miettiä valmiiksi jokin ratkaisu lokien kuluttamiseen, sillä tietojen onkiminen lukemalla jokaisen yksittäisen kontin lokeja erikseen on kuluttavaa ja aikaa vievää.

Toinen haaste, etenkin kehitysympäristöissä sekä jaetun palvelintilan (luku 2.6.1) ylläpitoratkaisuissa, on se että suorituskykyä tarvitaan enemmän kuin ilman konttien käyttöä. Tämä johtuu siitä, että koska jokainen kontti ajaa omaa versiotaan vaikkapa verkkopalvelimesta, kasvattaa tämä luonnollisesti esimerkiksi muistin käyttöä verrattuna malliin, jossa kaikki sivustot käyttäisivät samaa jaettua verkkopalvelinta.



Kuva 4.8: Latest-tagin käyttö voi aiheuttaa eroja levynkuvien versioissa.

Mikäli kontit tehdään käyttäen `latest`-tagia tai ilman lukittua versionumeroa (luku 3.3.2), saattaa tietokoneella tai palvelimella esiintyä useita eri versioita levynkuvasta (kuva 4.8). Tämä johtuu sekä konttien käynnistyksen ajankohdasta, että Dockerin välimuistista. Näin ollen sovelluksen julkaisuvaiheessa onkin tärkeää lukita tuotantoon vietävien Docker-levynkuvien versiot, jotta ympäristöjen välisiltä epäjohdonmukaisuuksilta voidaan välttyä ja voidaan toteuttaa Dockerin 'toimii samoin kaikkialla' –periaatetta. Tämä toki saattaa ajan kanssa kasvattaa esimerkiksi palvelimella sijaitsevien levynkuvien kokoa ja siten rajoittaa käytettävissä olevan kiintolevytilan määrää, mutta se on pieni hinta siitä, että sovellukset toimivat aina varmasti oikeassa versiossa.

4.4 Lopuksi

Dockerin vaikutuksia verkkosovellusten kehittämiseen ja ylläpitämiseen ei voida kiistää. Dockerista onkin tullut standardi kun puhutaan konttiteknologioista ja se saa tutkitusti

alati laajenevaa jalansijaa kehittäjien keskuudessa [35, 36, 37]. Seuraavassa luvussa käyn läpi kuinka Docker otetaan käyttöön pienessä verkkosivustoja ja -sovelluksia toteuttavassa ja ylläpitävässä yrityksessä.

Luku 5

Tapaustutkimus

5.1 Johdanto tapaustutkimukseen

Tapaustutkimuksen kohteena oli pieni yritys, joka suunnittelee ja toteuttaa verkkosivustoja sekä verkkopohjaisia sovelluksia. Tämä tutkimus keskittyy konttiteknologioiden käyttöönottoon kohdeyrityksessä, ja siihen miten tämä vaikutti työn tekemiseen sekä itse tekemisen näkökulmasta että työn laadullisesta näkökulmasta.

Työn tekemisen näkökulmalla tarkoitan sitä, että pystytäänkö konttiteknologioilla poistamaan työn kompleksisuutta, onko mahdollista poistaa joitakin työvaiheita tai onko mahdollista vähentää toistettavien työvaiheiden määrää tai sisältöä. Laadullisella näkökulmalla taas tarkoitan, voidaanko konttiteknologioilla parantaa työn tuloksen laatua eri näkökulmista kuten tietoturvan tai virhealttiuden kannalta.

Käyn ensin läpi kohdeyrityksen toimintatavat sekä käytössä olevat teknologiat ennen tutkimuksen aloittamista. Tämän jälkeen käyn läpi, miten konttiteknologia otettiin käyttöön yrityksen työskentelyn tueksi. Lopuksi käyn läpi miten onnistuneeksi toteutus katsottiin kun sitä vertaillaan aiempaan toimintatapaan ja prosesseihin aiemmin mainittujen metriikoiden näkökulmasta.

5.1.1 Käytetyt teknologiat ennen

Ennen konttiteknologioiden käyttöönottoa yrityksellä oli käytössään kaksi virtuaalipalvelinta, joissa verkkosivustoja ja sovelluksia ajettiin. Toinen palvelimista oli niin sanottu `staging`-palvelin, eli se oli tarkoitettu vain ja ainoastaan testauskäyttöön. Toinen palvelin taas oli `production`-palvelin, eli tällä palvelimella ajettiin tuotannossa olevia sivustoja ja sovelluksia. Kaikkien asiakkaiden sivustot siis sijaitsivat samalla palvelimella, mutta asiakkailla itsellään ei ollut näihin pääsyä, vaan heillä oli käytössään managed hosting -tyyppinen ylläpitosopimus. Kummassakin palvelimessa käyttöjärjestelmänä toimi CentOS Linux-jakelu. Jakelun olemattoman versiopäivityspolun takia palvelimet olivat eri versioissa; testipalvelin versiossa 6.5 ja tuotantopalvelin versiossa 7.

Kummastakin palvelimesta otettiin automaattiset varmuuskopiot päivittäin ja tämän lisäksi jokaisesta sivustosta otettiin erikseen varmuuskopiot myöskin päivittäin. Kaikki relevantit järjestelmäpäivitykset asennettiin kerran viikossa. Viimeisimpien kuukausien aikana tämä tosin tarkoitti testipalvelimen osalta vain tietoturvapäivityksiä tuen loputtua. Lisäksi sivustojen päivitykset asennettiin keskitetysti noin kerran viikossa.

Ohjelmistot kummallakin palvelimella olivat kutakuinkin yhdenmukaiset versionumeroita lukuun ottamatta ja tästä syystä seuraavassa kuvaamani rakenne pätee kumpaankin ympäristöön.

Tietoturva

Palvelimilla oli asennettuna useita tietoturvaa parantavia ohjelmistoja, kuten palomuuri. Tämän lisäksi kaikki tarpeettomat portit olivat suljettuja liikenteeltä ja SSH-yhteyksien ottamiseen käytössä ei ollut oletusporttia. Käytössä oli myös muutamia tietoturvaa parantavia ohjelmistoja, kuten `fail2ban`, joka skannaa palvelimen lokitiedostoja ja bannaa epäilyttävät IP-osoitteet. Perusteena sovellus käyttää muun muassa liian montaa kirjautumisyritystä väärällä salasanalla. Sovellusta käytettiin etenkin SSH-yhteyksien sekä `basic authentication` -tyyppisten kirjautumisten estämiseen. Lisäksi palvelimilla

oli asennettuna ja konfiguroituna `rkhunter`, joka etsi palvelimelta mahdollisia Rootkit-tyyppisiä haittaohjelmia.

Tietokanta

Tietokantoihin käytössämme oli avoimen lähdekoodin `MariaDB`-tietokantasovellus perinteisen MySQL:n sijaan. Koska MariaDB pohjautuu MySQL-tietokantasovellukseen, on se suoraan yhteensopiva esimerkiksi WordPress-julkaisujärjestelmän kanssa. Tämä oli meidän kannaltamme tärkeää, koska suurin osa verkkosivustoista on toteutettu WordPress:n kanssa. Jokaiselle sivustolle oli oma tietokantansa ja tähän tietokantaan luotiin sivustoa varten erillinen käyttäjätunnus, jolla oli oikeus käyttää kantaa ainoastaan lokaalisti. Näin mahdollisen hakkeroinnin sattuessa sivuston ja kannan yhdistävällä käyttäjällä ei olisi pääsyä muihin tietokantoihin.

WWW-palvelin

WWW-palvelimena yrityksellä oli käytössä Nginx. Nginx oli valikoitunut www-palvelimeksi etenkin sen erittäin alhaisen muistinkäytön että loistavien välimuistiominaisuuksien vuoksi. Koska Nginx ei pysty Apachen tapaan lataamaan moduuleita lennosta, oli käytössä suoraan lähdekoodista rakennettu versio. Näin buildiin mukaan saatiin halutut lisämoduulit `ngx_cache_purge` sekä `ngx_pagespeed`. Konfiguraationa käytimme pohjana h5bp:n luomaa pohjaa [49], josta muokkasimme omaan käyttöömme sopivan version [50]. Muokkaukset ottavat kantaa etenkin välimuistin toimintaan ja siihen, miten Nginx toimii yhteen WordPressin kanssa.

Koska suurin osa kohdeyrityksen tuottamista verkkosivuista ovat WordPress-pohjaisia, tarvitaan palvelimelta myös PHP-ohjelmointikieltä ja sen prosessointiin tarvittavia kirjastoja. Käytössämme oli PHP:n versio 7 ja prosessorina `php-fpm`, joka mahdollistaa muun muassa staattisen sisällön prosessoinnin ilman PHP-tulkin käyttöä.

5.1.2 Toimintatavat ennen

Kohdeyrityksessä käytössämme oli vakiintunut ja ennalta määritelty prosessi, jonka mukaan suurin osa projekteista toteutettiin. Prosessin osa-alueita toki kehitettiin aina kun se koettiin tarpeelliseksi, mutta runko oli melko vakiintunut.

Kehitysympäristö

Kehitystyö tapahtui kehittäjien työkoneilla, jotka olivat MacBook Pro -tietokoneita. Koska Macin OS X -käyttöjärjestelmä on Unix-pohjainen, toimii suurin osa tähän ympäristöön tehdyistä työkaluista ja sovelluksista, kuten `bash`, suoraan. Tämä helpottaa verkkosovellusten kehitystä, sillä sovelluksia ajetaan tuotannossa Linux-ympäristössä.

Kehitys aloitettiin kloonaamalla kohdeyrityksen kehittämä teemapohja nimeltä WPTB (WordPress Theme Boilerplate) kehittäjän omalle koneelle. Tämä pohja on vuosien varrella muodostunut kokonaisuus erilaisia asetuksia, lisäosia sekä komponenttipohjia, joita tarvitaan jokaisen sivuston luomisessa ja se toimiikin pohjana lähes jokaiselle luodulle WordPress-pohjaiselle sivustollemme.

Teemapohja konfiguroitiin projektin mukaiseksi, jonka jälkeen siitä tehtiin oma projektikohtainen repositorio versoinhallintaan. Versionhallintana käytössämme on Git ja etäpalvelimena tälle GitHub.

Kehitysympäristöissämme verkkopalvelimen roolia toimitti MAMP-niminen sovellus, jossa sisäänrakennettuina tulivat muun muassa Apache, Nginx, PHP sekä MySQL. Siis kaikki lokaalin palvelimen toiminnan kannalta relevantit sovellukset.

MAMPin avulla sovellukselle konfiguroitiin oma lokaali domain, josta kyseiseen projektiin pääsi käsiksi. Domainin juureen asennettin MAMPin avulla WordPress ja itse koodi sijoitettiin tämän juuren ulkopuolelle. Tämän jälkeen projektin teema sekä lisäosat linkattiin WordPress-asennukseen symbolisilla linkeillä. Tämä tehtiin, jotta WordPressin omat päivittyvät tiedostot eivät turhaan olisi mukana versionhallinnassa. Tässä esimerkki kansiorakenteesta:

```
my-project/ (.git)
    |
    |-- plugins/
    |-- themes/
        |
        |-- my-project-theme/
wordpress/
    |
    |-- (All WordPress files)
    |-- wp-content/
        |
        |-- plugins/    --> ../../my-project/plugins
        |-- themes/     --> ../../my-project/themes
        |-- uploads/
```

MAMPista pystyi helposti vaihtamaan esimerkiksi ajettavan PHP:n versiota, mutta tällöin tämä vaihtui globaalisti jokaiselle asennetulle sivustolle. MAMPin suurimpana ongelmana oli sen hankala konfiguroitavuus. Sivustoja oli todella haasteellista saada konfiguroitua samanlaisiksi kuin miten ne testiympäristössä tai tuotannossa toimivat. Lisäksi MAMPin päivitykset saattoivat usein rikkoa asetuksia ja toiminnallisuuksia, joten se toimi myös huomattavana epäluotettavuustekijänä kehittämiselle.

Testausympäristö

Testausympäristönä toimi jo aiemmin mainitsemani virtuaalipalvelin, jossa käyttöjärjestelmänä toimi CentOS 6.5. Itse sivusto pystytettiin ympäristöön tätä tehtävää varten luoduilla `bash`-scripteillä. Nämä funktiot loivat sivustolle tarvittavan hakemistorakenteen, nginx konfiguraation, DNS asetukset käyttäen Linoden rajapintoja, tietokannan sekä asensi WordPressin. Scriptille annettiin vain tulevan testisivuston domain ja scripti antoi syötteenä asennetun sivuston käyttäjätunnuksen ja salasanan. Scriptit varmensivat jokaisessa kohdassa edellisen suorituksen toimivuuden ja palasi alkuun, mikäli jokin kohta suorituksessa epäonnistui. Näin pystyimme välttämään virhetilanteita uusien ympäristöjen luon-

nissa. Lisäksi näillä samoilla scripteillä pystyi poistamaan jo olemassaolevia sivustoja sekä luomaan SSL-varmenteita sivustoille.

Tämän jälkeen koodirepositorioon ohjattu kustomoitu teema piti ohjata uuteen testisivustoon. Tämä tehtiin käyttämällä Deploybot-sovellusta. Deploybotin avulla pystyimme liittämään repositorion palveluun ja siirtämään koodin testipalvelimelle aina kun repositorio päivittyi. Tämän toiminnan pystyi käynnistämään joko automaattisesti jokaisesta päivityksestä tai manuaalisesti, jolloin Deploybotissa piti vielä lisäksi käydä painamassa siirto käyntiin. Testiympäristöissä meillä oli käytössä automaattinen siirto, jolloin jokainen `git push` –komento repositorioon päivitti automaattisesti testiympäristön.

Tuotantoympäristö

Tuotantoympäristö oli muutoin samanlainen kuin testiympäristö, mutta käyttöjärjestelmän versio oli uudempi, ja tästä syystä myös osa riippuvuuksista oli versioiltaan uudempia.

Tuotantopalvelimella käytössämme olivat samat ympäristöjen luontiskriptit sekä uuden koodin siirtäminen Deploybotin avulla. Deploybotista meillä oli kuitenkin käytössä manuaalinen siirto, joten siirrot tehtiin vasta kun ne oltiin ensin testattu toimiviksi testipalvelimella.

Sisällönsyöttö tuotantoon tehtiin joko alun alkaen käsin, tai sitten testipalvelimen sisältö kloonattiin tuotantopalvelimelle kun tämä otettiin käyttöön. Sekä kloonaukseen että sivustojen varmuuskopiointiin käytössämme oli ManageWP-palvelu. ManageWP hoiti myös varmuuskopiointia ja se teki testisivustoista varmuuskopiot kerran viikossa ja tuotantosivustoista päivittäin.

SSL-varmenteet tehtiin sivustoille usein manuaalisesti tuotantoympäristön luomisen jälkeen, ja nämä päivitettiin kaikkiin sivustoihin ajastetusti cronjobilla.

5.2 Konttiteknologioiden käyttöönotto

5.2.1 Suunnitelma

Konttiteknologioiden käyttöönotolla tarkoituksena oli saada Dockerin tuomat hyödyt liittyen etenkin sovellusten eristykseen toisistaan sekä siirrettävyyteen. Dockerin mahdollistama sovellusten eristäminen toisistaan oli erityisen tärkeää, sillä tarkoituksena oli pysyä mallissa, jossa samalla virtuaalipalvelimella suoritetaan edelleen useita eri sovelluksia. Näin pystyisimme hyödyntämään mikropalveluarkkitehtuuria (luku 4.2.1) kehityksessämme. Tämä mahdollistaa myös uusien palveluiden helpon lisäämisen sovellukseen jälkikäteen.

Tavoitteena oli luoda myös kohdeyritykselle standardoitu pohja sovellusten ajamiseen siten, että ne toimivat samoin sekä kehittäjän ympäristössä, testiympäristössä että tuotannossa. Lisäksi tarpeena oli luoda automatisoitu tapa luoda kokonaan uusia testi– ja tuotantopalvelimia.

Tämä oli myös hyvä tilaisuus arvioida vanhan systeemin toimivuutta sekä käytettyjen riippuvuuksien soveltuvuutta ja tarpeellisuutta. Sovimmekin kohdeyrityksen kanssa, että esimerkiksi Googlen PageSpeed-komponentista luovutaan sen jo aiemmin esiin tuomien vaikeasti selvitettävien ongelmien takia. Tärkeimpiä toiminnallisuuksia, kuten sivustoilla erinomaisesti toimivat välimuistiasetukset, oli tarkoitus säilyttää.

Yksi keskeinen kysymys käyttöönoton kannalta oli myös se, miten olemassa olevat sivustot saadaan kivuttomasti siirrettyä (*migrate*) uudelle pohjalle. Tästä syystä heti ensimmäisen prototyypin jälkeen tavoitteena olikin luoda uusi testipalvelin ja siirtää muutama toiminnassa oleva testiympäristö tänne ajoon. Lisäksi tehtävänäni oli kirjoittaa kohdeyritykselle dokumentaatio sivustojen siirrosta.

5.2.2 Toteutus

Olen jakanut toteutuksen kolmeen eri vaiheeseen. Ensin toteutin kohdeyritykselle pohjan, jonka avulla pystymme helposti luomaan uusia verkkosivupohjia, sekä käynnistämään olemassaolevia. Tämän jälkeen loin yritykselle Docker-pohjaisen välityspalvelimen HTTP-pyyntöjen ohjaamiseen oikeille konteille. Lopuksi loin Linodelle StackScriptin uusien palvelinten puoliautomaattiseen pystyttämiseen. Puoliautomaattisella tarkoitan sitä, että skriptin käyttö vaatii alkuasetusten syöttämisen manuaalisesti.

Sivuston luontipohja

Toteutin kohdeyritykselle `docker-compose`-tiedostoa hyödyntävän tuontipohjan WordPress-pohjaisten verkkosivustojen pystyttämiseen [51]. Yksittäinen projekti koostuu pohjassa kolmesta Docker-levynkuvasta. Ensimmäinen levynkuva on Nginx, joka toimii verkkopalvelimena ja vastaanottaa kaikki HTTP-pyynnöt. Nginx tarjoaa käyttäjälle suoraan kaikki staattiset tiedostot kuten HTML-tiedostot, kuvat, tyylitiedostot sekä JavaScript-tiedostot. Kun käyttäjä pyytää PHP-tiedostoja, ohjataan pyynnöt seuraavalle Docker-kontille, joka parsii nämä kävijälle. Tämä levynkuva sisältää myös WordPress asennuksen. Kolmas levynkuva on MariaDB, joka sisältää WordPressin tarvitseman tietokannan datan tallennusta varten.

Nginx levynkuva on tekemäni Ubuntuun pohjautuva levynkuva [52], johon asennetaan Ubuntun paketinhallinnasta `nginx-extras`-paketti [53]. Käytämme pakettia tavallisen `nginx`-paketin sijaan, koska se sisältää lisäosat välimuistin hallintaan ja sen poistamiseen. Levynkuvalle voidaan käynnistäessä antaa myös käyttäjä-id (*UID*), joka tällöin asetetaan nginx:ää ajavalle `www-data`-käyttäjälle. Tämä tulee tarpeeseen kun levynkuvaa ajetaan WordPress:in kanssa yhdessä ja välimuistitiedostot luodaan spesifillä käyttäjä-id:llä.

Myös WordPress:iä ajetaan käyttäen tekemääni Docker-levynkuvaa [54]. Levynkuva pohjautuu WordPressin viralliseen Alpine-linux käyttöjärjestelmästä luotuun Docker-

levynkuvaan. Levynkuvassa PHP:n versio on 7.1. Tämän levynkuvan päälle asennetaan muutamia tarpeellisia kirjastoja sekä WP-CLI -komentorivityökalu. Tämän jälkeen levynkuvalle määritellään räätälöidyt asetukset niin PHP:lle kuin php-fpm:lle. Lopuksi levynkuvan `ENTRYPOINT`-määreelle annetaan ajettavaksi skripti joka konfiguroi sähköpostiasetuksia sekä WordPressiin haluamiamme lisäasetuksia. Viimeisenä ennen käynnistystä konttia ajavan `www-data`-käyttäjän käyttäjä-id vaihdetaan vastaamaan isäntäkoneen käyttäjä-id:tä, jotta sieltä kiinnitettyihin kansioihin saadaan kirjoitusoikeus sekä kontin sisältä, että ulkoa isäntäkoneelta.

Luontipohjan kolmas sovellus on MariaDB-tietokanta, joka on levynkuvista ainoa, jossa on käytetty suoraan toimittajan virallista levynkuvaa. Tällekin sovellukselle lisäämme omat konfiguraatiot kiinnittämällä isäntäkoneella ja luontipohjan mukana tuleva konfiguraatiotiedosto konttiin. Itse tietokanta on kiinnitettynä myös siten, että sen sisältö tallennetaan isäntäkoneelle.

Loin luontipohjaan bash-scriptin, jonka avulla ladataan ympäristömuuttujat `.env`-tiedostosta, tarkistetaan että ne eivät ole tyhjiä ja luodaan `docker-compose`-tiedosto. Lopuksi skripti tarkistaa onko kontit jo käynnissä ja käynnistää ne mikäli näin ei ole. Mikäli kontit ovat jo käynnissä, skripti tyhjentää Nginx:n välimuistin. Tätä skriptiä kutsutaan joka kerta kun koodia viedään testaus- tai tuotantopalvelimelle, joten koodimuutosten tapahtuessa välimuisti tyhjennetään.

Compose-tiedostossa kontit liitetään toisiinsa ja niihin syötetään tarvittavat ympäristömuuttujat, kuten käytettävä domain, tietokannan salasana sekä muita tarvittavia tietoja ja asetuksia. Lisäksi kontit liitetään välityspalvelimeen ulkoisen Docker-verkon avulla.

Välityspalvelin

Aiemmassa ratkaisussamme palvelimillamme oli aioastaan yksi verkkopalvelin, joka ohjasi kaikki pyynnöt suoraan oikeille WordPress-asennuksille. Koska palvelimella vain yksi sovellus voi kerrallaan kuunnella yhtä porttia, tarvitaan HTTP- ja HTTPS-porttien

(80, 443) kuuntelemiseen välityspalvelinta, joka tekee tämän ohjauksen oikealle Docker-kontille. Näin ollen yhdenkään muun kontin ei tarvitse paljastaa porttejaan isäntäkoneelle tai ulkomaailmaan vaan kaikki liikenne ohjataan välityspalvelimen kautta.

Lisäksi, koska sivustojen piti toimia SSL-salattujen yhteyksien päällä, tuli välityspalvelimen tehtäväksi myös näiden sertifikaattien jakaminen ja uusiminen konteille Let's Encryptiä käyttäen.

Koostin ja konfiguroin valmiista Docker-levynkuvista kohdeyritykselle välityspalvelimen [55], joka kuuntelee samaan Docker-verkkoon `nginx-proxy-network` liittyneitä kontteja, luo niille Nginx-konfiguraatiot ohjaamaan liikenteen oikeaan konttiin sekä luo konteille SSL-sertifikaatit.

Välityspalvelin koostuu kolmesta erillisestä Docker-kontista: ensimmäinen on normaali Nginx-palvelin, joka kuuntelee portteja 80 sekä 443. Seuraava kontti kuuntelee Dockerin sokettia ja generoi Nginx:lle uuden konfiguraation aina kun samaan verkkoon kytkeytynyt kontti sammuu tai käynnistyy. Kolmas kontti luo käynnistyneille ja konfiguroiduille konteille sertifikaatit salattuja yhteyksiä varten sekä uusii nämä aika ajoin.

Dockerin sokettia kuunteleva kontti ei kuitenkaan luo konfiguraatioita jokaiselle käynnistetylle kontille, vaan se valitsee vain kontit, joille käynnistettäessä osoitetaan jokin domain ympäristömuuttujan `VIRTUAL_HOST` avulla. Sama pätee myös sertifikaatteja luovaan konttiin. Se luo sertifikaatin vain konteille, jotka käynnistetään ympäristömuuttujilla `LETSENCRYPT_HOST` sekä `LETSENCRYPT_EMAIL`. Näin ollen konfiguraatiot saadaan luotua vain halutuille konteille.

Mikäli toteutuksen palvelinratkaisuksi olisi valittu jokin PaaS-pilvipalveluiden tarjoaja (luku 4.2.2), ei omaa välityspalvelinta tarvittaisi, sillä tämä ominaisuus sisältyy palveluiden tarjontaan.

Linode StackScript

Koska kohdeyritys päätti jatkaa Linoden tarjoamien virtuaalipalvelinten käytössä ja tavoitteena oli luoda tapa tuottaa uusia testi- ja tuotantopalvelimia helposti, oli tähän helppona valintana Linoden tarjoama StackSript-palvelu. Ideana on, että kun uutta palvelinta luodaan, valitaan esivalittu skripti, jonka avulla palvelin esikonfiguroidaan.

Loin kohdeyritykselle skriptin (liite A), jonka avulla tällaisia palvelimia voi helposti luoda. Tämä skripti pyytää aluksi käyttäjältä tietoja, kuten perustettavan palvelimen isäntänimeä, luotavan käyttäjän käyttäjänimeä, salasanaa ja julkista SSH-avainta sekä palvelimelle asetettavaa aikavyöhykettä. Pyydettävät kentät määritellään tämän `bash`-skriptin kommenteissa Linoden määrittelemällä notaatiolla [56].

Tämän jälkeen skripti siirretään palvelimelle ja ajetaan. Ajettaessa se mm. asettaa palvelimelle isäntänimen sekä aikavyöhykkeen, luo palvelimelle uuden käyttäjän ja mahdollistaa kirjautumisen annetulla SSH-avaimella, asentaa dockerin, docker-composen sekä muita vaadittavia kirjastoja sekä konfiguroi palomuurin.

Näillä asetuksilla kohdeyritys voi helposti luoda uusia virtuaalipalvelimia käyttöönsä ja siirtää ja käynnistää Docker-kontteja palvelimelle ajettavaksi. Tätä skriptiä käyttämällä loimme kohdeyritykselle alkuun testipalvelimen ja myöhemmin myös tuotantopalvelimen.

5.2.3 Analyysi

Saavutukset

Toteutuksen tarkoituksena oli luoda kohdeyritykselle standardoitu pohja sovelluskehityksen tueksi siten, että se käyttää hyväkseen Dockerin tuomia mahdollisuuksia (luku 4). Toteutus jakaa yksittäisen verkkosovelluksen eri osa-alueet, kuten tietokannan ja verkkopalvelimen, omiin mikropalveluihinsa (luku 4.2.1). Näin jokainen sovellus toimii toisistaan riippumatta. Ainoa ylemmän tason riippuvuus palveluille on itse Dockerin lisäksi toi-

miva välityspalvelin (luku 5.2.2), jonka avulla kaikki HTTP-pyynnöt välitetään. Lisäksi pohja mahdollistaa jatkuvan integraation (luku 4.2.3) esimerkiksi siten, että yksittäiseen sovellukseen voidaan helposti liittää erillinen kontti ajamaan testejä tai tekemään tuotantotiedostoja jokaisen versionhallintaan pusketun muutoksen jälkeen. Pohja myös täyttää yhden kohdeyritykselle tärkeimmistä kriteeteistä, sillä sen avulla saadaan aikaiseksi riippumattomuus kehittäjän erikseen asennettavista sovelluksista sekä pienillä modifikaatioilla jopa kehittäjän käyttöjärjestelmästä. Näin se toimii erittäin hyvänä ja laajennettavana työkaluna kehitystyön apuna (luku 4.2.5).

Sovelluspohjan avulla saavutettiin kaikki tapaustutkimukselle asetetut vaatimukset ja työn edetessä pohjan avulla mahdollistettiin myös asioita, jotka eivät olleet vaatimuslistalla, mutta nousivat esiin hyvinä kehityskohteina. Tällaisia oli muun muassa muutamiin testiympäristöihin rakennetut automatiikat. Kehitys vaati myös versionhallinnan haaroittamista (*git branching*), jotta jo olemassa olevat sovellukset voitiin muuttaa yhteensopiviksi pohjan kanssa. Tästä saatiin uusi työskentelytapa myös kohdeyrityksen muuhun sovelluskehitykseen.

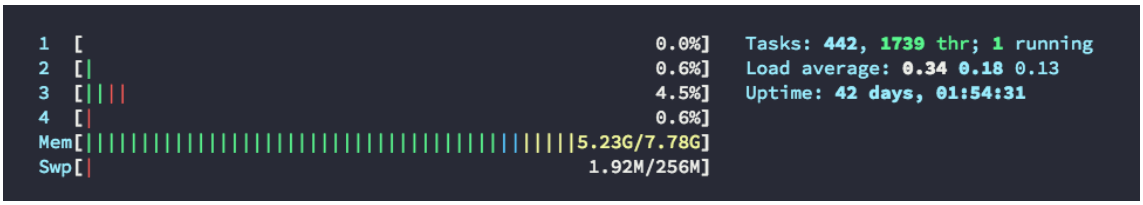
Sovelluspohjan ja siihen liittyvän välityspalvelimen avulla myös SSL-varmenteiden saaminen sivustoille ja sovelluksille on uudella järjestelmällä täysin automaattinen. Testipalvelimella käytössämme on ns. wildcard-domainit, joten varmenteet saadaan uusille sivuille vieläpä ilman DNS-asetusten päivittymisen tuomaa odottelua.

Haasteet

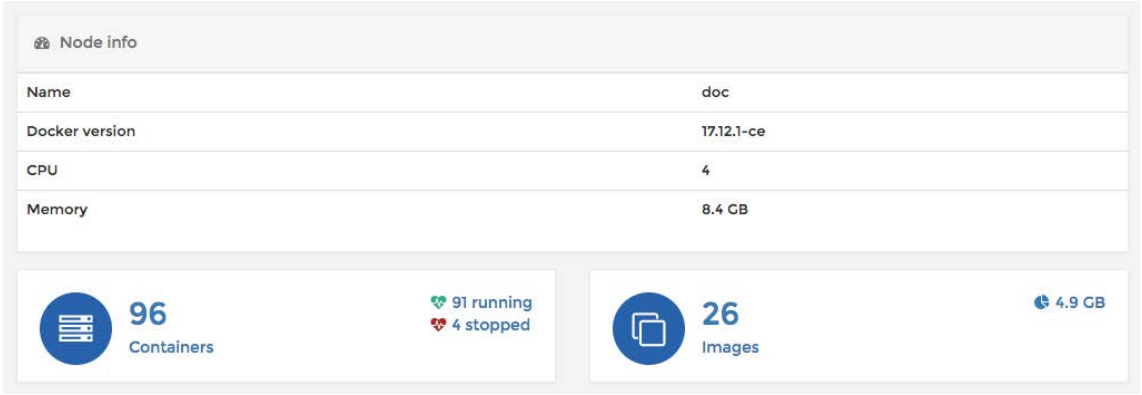
Suurimpana haasteena koin Dockerin volumet ja eritoten tiedostojen luku- ja kirjoitusoikeudet näiden kanssa työskennellessä. Ongelmana on se, että kun sovellukseen kiinnitetään kansioita isäntäkoneelta, on näillä tiedostoilla eri omistajat kuin kontin sisällä (luku 3.2.2). Ratkaisin ongelman osittain syöttämällä konteille isäntäkoneen käyttäjän käyttäjä-id:n, jonka jälkeen kontin käyttäjän käyttäjä-id muutettiin vastaamaan tätä. Tämä johti ongelmaan, joka on ilmenee eritoten Dockerin Mac OSX-käyttöjärjestelmän le-

vyjärjestelmän kanssa. Kirjoittaminen Dockerin volumen sisään on erittäin hidasta[1].

Toinen haaste liittyy muistin käyttöön. Esimerkiksi testipalvelimellamme on tällä hetkellä vajaa sata konttia käynnissä ja näistä jokainen vielä oman osansa muistista, vaikka sovellus ei olisi käytössä (kuvat 5.1 ja 5.2). Teoreettinen ratkaisu tähän olisi sammuttaa käyttämättömät kontit tietyn ajan jälkeen ja käynnistää ne taas uuden HTTP-pyynnön saapuessa. Tähän ei kuitenkaan löytynyt olemassa olevia sekä helposti lähestyttäviä malleja.



Kuva 5.1: Docker-kontit vievät paljon muistia myös tyhjäkäynnillä



Kuva 5.2: Testipalvelimella käynnissä yli 90 konttia

Saimme optimoitua muistinkäyttöä säätämällä tietokantojen muistinkäyttöasetuksia sekä muokkaamalla `php-fpm daemon`:it käynnistymään automaattisesti vain tarvittaessa. Lisäksi `nginx`:n toimiva välimuisti mahdollistaa sen, että suuren osan kutsuista ei koskaan tarvitse mennä PHP:n suoritettavaksi, vaan sivut voidaan tarjota välimuistista.

[1]https://docs.docker.com/docker-for-mac/osxfs/#performance-issues-solutions-and-roadmap

Muuta

Tapaustutkimuksessa ei vielä päästy testaamaan Dockerin mahdollisuuksia sovellusten skaalaamisen osalta (luku 4.2.2). Tämä johtuu osittain tarpeen puutteesta, sekä valitusta ylläpitoratkaisusta. Sovellusten skaalautuminen on kuitenkin mielenkiintoinen jatkokehitysmahdollisuus.

Toinen mielenkiintoinen ja jatkokehitystä vaativa ongelma on Nodea vaativien työkalujen Dockeroiminen. Tämä tarkoittaisi sitä, että kehittäjällä ei tarvitsisi olla Nodea asennettuna omalle työasemalleen, vaan myös JavaScript- ja CSS-tiedostojen kääntäminen tapahtuisi konteissa. Testasin tätä jo nopeasti (koodiesimerkki 5.1) ja se toimii muuten hienosti, mutta selaimen automaattinen päivittäminen käyttäen `browser-sync`–sovellusta vaatii vielä tutkimista ja lisäkonfiguraatiota kutsujen välittämiselle palvelulta toiselle.

```
1 docker run \
2   -u node \
3   -w /home/node/app \
4   -v $PWD/<path-to-src-files>:/home/node/app \
5   --rm \
6   node:9.5.0 \
7   bash -c 'yarn --non-interactive && yarn build'
```

Koodiesimerkki 5.1: Docker-levynkuvalla voidaan helposti ajaa yksittäinen komento esimerkiksi tuotantotiedostojen rakentamiseen

Eräs toteutusta tehdessä tullut idea oli, että mikäli sovellus tarvitsee esimerkiksi oman FTP/SFTP-palvelimen, voidaan tämä kiinnittää sovelluskokonaisuuteen omana konttinaan. Selvittelyiden jälkeen ilmeni kuitenkin, että kyseiset protokollat eivät tue yhdistämistä pelkän domain-nimen avulla, joten liikenteen ohjaaminen oikeaan konttiin ei olisi mahdollista. Tähän ratkaisuksi teimme näille palveluille oman palvelimen, jossa vangitaan käyttäjät omiin kotikansioihin ja liikenne sovelluksen ja siihen liittyvän SFTP-

palvelimen välillä ratkaistiin erillisellä kontilla, joka tarjoaa `htpasswd`-autentikoinnin palvelimelle.

5.3 Dockerin vaikutukset yrityksen kehitysprosessiin

Vaikka Dockerilla ei sinäänsä ole suoria teknologisia vaikutuksia kohdeyrityksen sovelluskehitysprosessin ensimmäisiin vaiheisiin, eli sovelluksen määrittelyyn ja suunnitteluun (luku 4.3.1), huomasimme sen vaikuttavan joiltakin osin määrittelyvaiheessa tehtäviin ratkaisuihin esimerkiksi teknologiavalintojen osalta. Sovelluksia on entistä helpompi suunnitella mikropalveluarkkitehtuurin mukaisesti toimiviksi. Lisäksi huomasimme että koska Docker nopeuttaa hieman etenkin kehitystyön ensimmäisiä vaiheita eli ympäristöjen perustamisia, tätä aikaa voidaan tulevaisuudessa resursoida esimerkiksi määrittelyyn tai suunnitteluun.

Suurimmat vaikutukset kohdeyrityksen työskentelyyn uusi pohja toi eritoten kehitystyön tekemiseen, sekä sovellusten ylläpidon järjestämiseen (luvut 4.3.2 ja 4.3.4). Helpoiten huomattavia ajallisia vaikutuksia huomasimme projektissa tarvittavien ympäristöjen luomisen yhteydessä. Uudella pohjalla uuden projektin aloittaminen onnistuu muutamissa minuuteissa ja projekti on erittäin helppoa esimerkiksi kopioida myöhemmin mukaan tulevalle kehittäjälle. Lisäksi pohja tuo suuremman varmuuden toimivuuden takaamiseksi myös tuotannossa, sillä sivustoja ylläpidetään saman pohjan avulla eroina vain pieniä konfiguraatiomuutoksia, jotka nekin on naurettavan helppo testata ensin lokaalisti.

Näkisin suurimman, vielä hyödyntämättömän, potentiaalin olevan sovellusten testaamisessa erilaisissa ympäristöissä Dockerin avulla (luku 4.3.3). Tätä en kuitenkaan vielä ole päässyt täysin hyödyntämään, mutta se on erittäin kiinnostava jatkokehityskohde.

Kollegoideni mielipiteet työni tuloksista ovat linjassa omien johtopäätösteni kanssa ja he ovat nostaneet samoja hyötynäkökulmia. Dockerin käyttöönotto kohdeyrityksessä vaikutti positiivisesti sekä alussa mainitsemani työn tekemisen näkökulmasta, että etenkin

työn laadullisesta näkökulmasta. Kohtalaisen matalan oppimiskäyrän jälkeen työvaiheiden määrä väheni etenkin aloitettaessa uusia projekteja. Myös työn tulosten laatu tuntui parantuvan ja etenkin tuotantoversion käyttäytymisen ennustettavuus toi lisää turvallisuudentunnetta. Myös työskentelymukavuus Dockerin kanssa on tuntunut aiempiin toimintatapoihin verrattuna paremmalta. Tämä saattaa toki johtua vain uutuudenviehätyksestä.

Kaiken kaikkiaan voisin siis sanoa, että Dockerin soveltaminen yrityksen sovelluskehitysprosessiin on tuonut useita hyötyjä ja mahdollisuuksia, joista kaikkia emme ole vielä edes tunnistaneet.

Luku 6

Johtopäätökset

Docker toimii erittäin hyvin sekä apuvälineenä kehitystyössä että alustana vaativammallekin sovelluskehitykselle ja näiden ylläpidolle. Sen avulla voidaan helposti luoda niin pohjia erilaisille sovelluksille, testata nopeasti ohjelmointikirjastoja ja näiden useita eri versioita sekä rakentaa vakaa ja tuotantokelpoinen ylläpitojärjestelmä Dockeriin pohjautuen.

Ensin kävin toisessa luvussa läpi millaisista työvaiheista verkkosovelluksen kehityskaari koostuu pohjautuen niin työn kohdeyrityksen toimintatapoihin, kuin alan yleisiin toimintatapoihin. Kävin läpi millaisia asioita kukin työvaihe pitää sisällään, jotta voin myöhemmin tutkia konttiteknologioiden vaikutuksia näihin.

Kolmannessa luvussa kävin läpi konttiteknologioiden periaatetta ja toiminnallista pohjaa. Tunnistin myös muutamia mahdollisia teknologiaan liittyviä ongelmia. Lopuksi syvennyin tällä hetkellä käytetyimpään ja suosituimpaan konttiteknologioiden sovellukseen Dockeriin. Kävin läpi, millaisista rakennuspalikoista Docker koostuu ja muutaman esimerkin kautta kuinka sitä voidaan käyttää.

Neljännessä luvussa tutkin kuinka Docker toimii sovelluskehityksen tukena ja millä eri kehitysalueilla siitä on mahdollisesti hyötyä ja mitä kehitystapoja se teknologiana tukee. Löysin Dockerille niin hyödyllisiä käyttökohteita kuin erilaisia tapoja soveltaa Dockeria sovelluskehityksen tukena.

Viidennessä luvussa kävin läpi kohdeyritykselle toteuttamani sovelluskehityspohjan, jonka lähtökohtana oli niin uusien projektien helppo käynnistäminen kuin ylläpitokin. Hyödynsin tätä tehdessä neljännen luvun löydöksiäni ja sain koostettua toimivan mikropalveluarkkitehtuuria tukevan pohjan, jonka avulla projektin toistettavuus eri ympäristöissä on tehokasta ja projektien siirrettävyys parantui aiempaan tapaan nähden huomattavasti.

Vaikka Docker onkin saavuttanut jo melko hyvän vakauden, on sen käytössä vielä ongelmia, etenkin yhteensopivuus Mac OSX-käyttöjärjestelmän levyjärjestelmän kanssa vaatii vielä kypsymistä. Tämä ongelma ei kuitenkaan rajoita teknologian käyttöä vaan toimii lähinnä pienenä hidasteena. Muutamista tutkimuksen aikana kohtamistani ongelmista huolimatta Docker osoittautui erittäin käyttökelpoiseksi tavaksi tukea pienen yrityksen verkkosovelluskehitystä ja konttiteknologiat mahdollistavat sovellusten ylläpidon ilman tätä tehtävää varten dedikoitua henkilöä kaventaen tarvetta erilliselle ylläpito-osaajalle. Kaiken kaikkiaan voinkin siis sanoa, että Docker on erinomainen teknologia osana verkkosivustojen kehittämistä sekä niiden ylläpitoa.

Lähteet

[1] Total number of Websites. [Online]. Available: http://www.internetlivestats.com/total-number-of-websites/

[2] M. Taylor, J. McWilliam, H. Forsyth, and S. Wade, "Methodologies and website development: a survey of practice," *Information and Software Technology*, vol. 44, no. 6, pp. 381 – 391, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584902000241

[3] D. Turk, R. B. France, and B. Rumpe, "Limitations of agile software processes," *CoRR*, vol. abs/1409.6600, 2014. [Online]. Available: http://arxiv.org/abs/1409.6600

[4] A. Hussain and E. O. C. Mkpojiogu, "Requirements: Towards an understanding on why software projects fail," *AIP Conference Proceedings*, vol. 1761, no. 1, 2016. [Online]. Available: https://aip.scitation.org/doi/abs/10.1063/1.4960886

[5] J. Stevens. (2016) Different types of web hosting. [Online]. Available: https://hostingfacts.com/different-types-of-web-hosting/

[6] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, vol. 25482, pp. 171–172, 2015. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=7095802

[7] chroot(8) - OpenBSD manual pages. [Online]. Available: http://man.openbsd.org/chroot

[8] P. Hope, "Using Jails in FreeBSD for fun and profit," *Login; The Magazine of Usenix & Sage*, vol. 27, no. 3, pp. 48–55, 2002. [Online]. Available: https://www.usenix.org/system/files/login/articles/1296-hope.pdf

[9] P.-H. Kamp and R. N. M. Watson. (2000) Jails: Confining the Omnipotent Root. [Online]. Available: https://docs.freebsd.org/44doc/papers/jail/jail.html

[10] J. Gélinas, "vserver 0.0 changes log." [Online]. Available: http://www.solucorp.qc.ca/changes.hc?projet=vserver&version=0.0

[11] Thildred, "The History of Containers – Red Hat Enterprise Linux Blog," 2015. [Online]. Available: http://rhelblog.redhat.com/2015/08/28/the-history-of-containers/

[12] P. B. Menage, "Adding Generic Process Containers to the Linux Kernel," *Proceedings of the Ottawa Linux Symposium*, pp. 45–58, 2007. [Online]. Available: https://www.kernel.org/doc/ols/2007/ols2007v2-pages-45-58.pdf

[13] M. Ridwan. (2015) Namespaces Tutorial: Isolate Your Linux System — Toptal. [Online]. Available: https://www.toptal.com/linux/separation-anxiety-isolating-your-system-with-linux-namespaces

[14] Linuxcontainers.org. Linux Containers - LXC - Introduction. [Online]. Available: https://linuxcontainers.org/lxc/introduction/

[15] A. Avram, "Docker: Automated and Consistent Software Deployments," 2013. [Online]. Available: https://www.infoq.com/news/2013/03/Docker

[16] D. Ernst, D. Bermbach, and S. Tai. Understanding the Container Ecosystem : A Taxonomy of Building Blocks for Container Lifecycle and Cluster Management.

[Online]. Available: http://www.ise.tu-berlin.de/fileadmin/fg308/publications/2016/Ernst_WoC_container-taxonomy.pdf

[17] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," in *Present and Ulterior Software Engineering*, 2016, pp. 195–216. [Online]. Available: http://arxiv.org/abs/1606.04036

[18] J. Chelladhurai, P. R. Chelliah, and S. A. Kumar, "Securing docker containers from Denial of Service (DoS) attacks," *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, no. October 2017, pp. 856–859, 2016.

[19] F. Kung. (2014) Memory inside Linux containers. [Online]. Available: https://fabiokung.com/2014/03/13/memory-inside-linux-containers/

[20] D. Bertović. (2016) Handling Permissions with Docker Volumes. [Online]. Available: https://denibertovic.com/posts/handling-permissions-with-docker-volumes/

[21] J. Pitt. (2016) Docker.qcow2 never shrinks - disk space usage leak in docker for mac #371. [Online]. Available: https://github.com/docker/for-mac/issues/371

[22] A. Hawkins. (2016) Container technologies: more than just Docker. [Online]. Available: https://cloudacademy.com/blog/container-technologies-more-than-dockers/

[23] L. Stamey. Docker's Tools of Mass Innovation: Explosive Growth From Open-Source Containers to Commercial Platform for Modernizing and Managing Apps - HostingAdvice.com. [Online]. Available: http://www.hostingadvice.com/blog/dockers-explosive-growth-from-open-source-containers-to-commercial-platform/

[24] L. Carlson. (2014) What is Docker and When to Use It. [Online]. Available: https://www.ctl.io/developers/blog/post/what-is-docker-and-when-to-use-it/

[25] Docker Inc. About Docker Engine. [Online]. Available: https://docs.docker.com/engine/

[26] ——. About images, containers, and storage drivers. [Online]. Available: https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/

[27] ——. Docker Overview. [Online]. Available: https://docs.docker.com/engine/docker-overview/

[28] ——. Docker Volumes. [Online]. Available: https://docs.docker.com/engine/admin/volumes/volumes/

[29] ——. Docker container networking. [Online]. Available: https://docs.docker.com/engine/userguide/networking/

[30] P. Raj, J. Chelladhurai, and V. Singh, *Learning Docker*. Packt Publishing, 2015. [Online]. Available: https://books.google.fi/books?id=jkkOCgAAQBAJ

[31] Docker Inc. Overview of Docker Compose. [Online]. Available: https://docs.docker.com/compose/overview/

[32] ——. Docker Registry. [Online]. Available: https://docs.docker.com/registry/

[33] ——. Docker Cloud. [Online]. Available: https://docs.docker.com/docker-cloud/

[34] Get Docker CE for Ubuntu. [Online]. Available: https://docs.docker.com/engine/installation/linux/docker-ce/ubuntu/

[35] RightScale. (2015) State of the Cloud Report. [Online]. Available: https://assets.rightscale.com/uploads/pdfs/RightScale-2015-State-of-the-Cloud-Report.pdf

[36] ——. (2016) State of the Cloud Report. [Online]. Available: https://assets.rightscale.com/uploads/pdfs/RightScale-2016-State-of-the-Cloud-Report.pdf

[37] ——. (2017) State of the Cloud Report. [Online]. Available: https://assets.rightscale.com/uploads/pdfs/RightScale-2017-State-of-the-Cloud-Report.pdf

[38] ——. (2016) State of the Cloud Report: DevOps Trends. [Online]. Available: http://assets.rightscale.com/uploads/pdfs/rightscale-2016-state-of-the-cloud-report-devops-trends.pdf

[39] Docker Inc. (2016) Evolution of the Modern Software Supply Chain - Docker Survey. [Online]. Available: https://goto.docker.com/rs/929-FJL-178/images/Docker-Survey-2016.pdf

[40] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The making of cloud applications: an empirical study on software development for the cloud," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, 2015, pp. 393–403. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2786805.2786826

[41] H. Harms, C. Rogowski, and L. Lo Iacono, "Guidelines for adopting frontend architectures and patterns in microservices-based systems," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pp. 902–907, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3106237.3117775

[42] D. Namiot and M. Sneps-sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[43] M. Villamizar, O. Garces, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang, "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," *Proceedings - 2016 16th IEEE/ACM International*

Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016, pp. 179–182, 2016.

[44] Google. An update on container support on Google Cloud Platform. [Online]. Available: https://cloudplatform.googleblog.com/2014/06/an-update-on-container-support-on-google-cloud-platform.html

[45] Netflix. The Evolution of Container Usage at Netflix. [Online]. Available: https://medium.com/netflix-techblog/the-evolution-of-container-usage-at-netflix-3abfc096781b

[46] A. Larson. (2017) How Docker Containers Are Being Used Right Now. [Online]. Available: https://blog.newrelic.com/2017/04/18/docker-usage-dockercon-devops/

[47] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, May 2014.

[48] Matt Manning. (2015) A Tale of Two Onboardings. [Online]. Available: https://convox.com/blog/a-tale-of-two-onboardings/

[49] H5bp – Nginx Server Configs. [Online]. Available: https://github.com/h5bp/server-configs-nginx

[50] Bond Agency. Nginx server configs. [Online]. Available: https://github.com/bond-agency/server-configs-nginx

[51] Bond Agency and R. Merikukka. docker-wpbp. [Online]. Available: https://github.com/bond-agency/docker-wpbp

[52] R. Merikukka. nginx-extras. [Online]. Available: https://github.com/roopemerikukka/nginx-extras

[53] [Online]. Available: https://packages.ubuntu.com/xenial/nginx-extras

[54] Bond Agency and R. Merikukka. docker-wpbp-wordpress. [Online]. Available: https://github.com/bond-agency/docker-wpbp-wordpress

[55] ——. nginx-le-proxy. [Online]. Available: https://github.com/bond-agency/nginx-le-proxy

[56] R. Feliciano. (2014) Automate Deployment with StackScripts. [Online]. Available: https://www.linode.com/docs/platform/stackscripts/

Liite A

Linode StackScript

```
#!/bin/bash

# <UDF name="HOSTNAME" label="Select machine hostname" />
# <UDF name="FQDN" label="Give your FQDN" />
# <UDF name="TZ" label="Your Timezone" default="Europe/Helsinki" />
# <UDF name="USERNAME" label="New users username" default="bond" />
# <UDF name="PASSWORD" label="New users password" />
# <UDF name="USERPUBKEY" label="New users public SSH key" />

set -e
set -u

# Logs for StackScript
exec &> /root/stackscript.log

# Include Linode StackScript Bash Library
source <ssinclude StackScriptID=1>

# Disable all interactive parts from installations
export DEBIAN_FRONTEND=noninteractive

system_update

# Add user
user_add_sudo $USERNAME $PASSWORD

# Add public key for user
user_add_pubkey "$USERNAME" "$USERPUBKEY"
chmod 700 -R /home/$USERNAME/.ssh

```

```
31 # Disables root SSH access and password login.
32 sed -i 's/PermitRootLogin yes/PermitRootLogin no/' \
33     /etc/ssh/sshd_config
34 sed -i 's/PasswordAuthentication yes/PasswordAuthentication no/' \
35     /etc/ssh/sshd_config
36 sudo systemctl restart sshd
37
38 # Get the machines IP address
39 IPV4_ADDR=$(/sbin/ifconfig eth0 | awk '/inet / { print $2 }' | \
40     sed 's/addr://')
41 IPV6_ADDR=$(/sbin/ifconfig eth0 | grep 'inet6' \ |
42     grep -i 'global' | awk -F ' ' '{print $3}' \ |
43     awk -F '/64' '{print $1}')
44
45 # Set hostname
46 hostnamectl set-hostname $HOSTNAME
47
48 # Update /etc/hosts
49 echo "${IPV4_ADDR} ${FQDN} ${HOSTNAME}" >> /etc/hosts
50 echo "${IPV6_ADDR} ${FQDN} ${HOSTNAME}" >> /etc/hosts
51
52 # Set Timezone
53 timedatectl set-timezone $TZ
54
55 # Install dependencies
56 apt-get install -y \
57     apt-transport-https \
58     ca-certificates \
59     curl \
60     software-properties-common \
61     python-pip \
62     htop \
63     wget \
64     sendmail \
65     ufw \
66     unzip \
67     curl
68
69 # Add Docker's official GPG key
70 curl -fsSL https://download.docker.com/linux/ubuntu/gpg | \
71     sudo apt-key add -
72
73 # Install latest stable version of Docker
74 add-apt-repository \
75    "deb [arch=amd64] https://download.docker.com/linux/ubuntu \
76    $(lsb_release -cs) \
77    stable"
78
```

```
79  apt-get update && apt-get install -y docker-ce
80
81  # Update pip
82  pip install --upgrade pip
83
84  # Install Docker Compose
85  pip install -U docker-compose
86
87  # Configure Docker to start on boot
88  systemctl enable docker
89
90  # Add the new user to docker group
91  usermod -aG docker $USERNAME
92
93  # Setup UFW
94  ufw default allow outgoing
95  ufw default deny incoming
96  ufw allow 22
97  ufw allow 80
98  ufw allow 443
99  ufw enable
100
101 # Export user id of the $USERNAME
102 echo "export USER_ID=\$(id -u)" >> /home/$USERNAME/.bashrc
```